Thesis Project for the degree of

Master of Science

in Computer Science

Implementation of a Parser Generator

Pace University

School of Computer Science and Information Systems

Josh Foure

Professor Badii - Thesis Advisor

Spring 2004 - Fall 2004

Pace University

Date: 12/06/2004

Student's Name: Josh Foure

Date of Graduation: Spring 2005

Title of Thesis: Implementation of a Parser Generator

This will certify that the above thesis has been approved in fulfillment of the thesis option for the Degree of Master of Science in Computer Science.

Student's Signature _____

Thesis Advisor _____

Committee Member _____

Committee Member _____

Department Chair _____

Dean _____

Date _____

Abstract

A grammar can be expressed in EBNF format to define its production rules. Given such an EBNF syntax, one can systematically apply rules to create a simpler BNF with which a computer can more easily work. From the simpler BNF, AST objects can be derived and a scanner and parser created. The Parser Generator program I have implemented takes 2 inputs: a grammar expressed in EBNF and a list of tokens expressed in a proprietary format. The output is two programs, a scanner and a parser, as well as many supporting classes than can scan and parse any source code written in the language of the EBNF. This paper explains the transformations applied to the inputs as well as the algorithms used to generate a scanner and parser.

Table of Contents

Introduction to Languages

In order to communicate with someone, you must agree on a common language. If you are writing a letter to someone in France you have to decide if you will be writing in English or in French. If you write "I see the mean dog" in English but the person to whom you are writing expected you to write in French, they will not understand what you are saying. They will recognize all the letters but they won't understand the meaning. Similarly, if they were to write to you in French "Je vois le méchant chient" you may not even recognize all the letters because the "é" is not an english letter. If you agree that you are going to use English words, you still need to make sure to use the correct ordering. For example, "I see dog the mean" makes no sense.

A scanner is a program that reads characters and plays 2 roles: it makes sure that each character entered is allowed (e.g. no "é" in English words) and it groups characters together in logical groupings called tokens. Tokens in our English language example would be things like word tokens and punctuation mark tokens such as comma tokens and question mark tokens. A parser is program that reads in tokens (provided by a scanner) and validates that the tokens are in a correct order (e.g. that the sentence is not "I see dog the mean"). In the English language, for example, there is a rule that after a comma token, you can not have a period token. This is something that would be validated by the parser. The parser also generates an object tree that represents the token entered in memory.

Humans are very good at seeing patterns in words. In fact, a few years ago there was an email hoax on the internet that suggested the following:

> "Aoccdrnig to a rscheearch at Cmabrigde Uinervtisy, it deosn't mttaer in waht oredr the ltteers in a wrod are, the olny iprmoetnt tihng is taht the frist and lsat ltteer be at the rghit pclae. The rset can be a toatl mses and you can sitll raed it wouthit porbelm. Tihs is bcuseae the huamn mnid deos not raed ervey lteter by istlef, but the wrod as a wlohe."[i]

If you are having problems reading this, it says that "According to a researcher at Cambridge University, it doesn't matter in what order the letters in a word are, the only important thing is that the first and last letter be at the right place. The rest can be a total mess and you can still read it without problem. This is because the human mind does not read every letter by itself but the word as a whole." It turns out that this research was not actually conducted at Cambridge University but it does get the point across. A computer would never be able to understand that sentence because even though it recognizes the letters themselves, the words make no sense. In fact, Microsoft Word which was used to type this is paper highlighted the majority of the words because it does not recognize them in its spellchecker dictionary.

Characters

A character is the building block of a language. A character is atomical in the sense that it doesn't male sense to talk about a half character or to break it up in any way. Most characters can be typed using a regular keyboard and have a visual display. For example, in the English language the symbols are all the lower case letters, the upper case letters, the numbers, and every other special character such as a hyphen or parenthesis. Even though you can type a pound sign "#" on an English keyboard, you can not have a pound character in an English word. Different languages define the allowable characters they accept.

Some characters are also designated as delimiters between characters that that up tokens. For example, in the English language we usually have a space between each word. Additionally, indentation and line breaks also have no impact on the meaning of a sentence. However, one language's separator is another language's data character. There is a language called Whitepace[11] that consists only of non-visible characters such as space, line feed and tab.

We will refer to a document that consists of zero or more characters that will be scanned and parsed as a "source code". Although this term is borrowed from the programming language field, where programmers write source code that is then compiled into another form (e.g. binary or byte codes), we use the term "source code" much more loosely. As we have seen when discussing the English language, a scanner and parser would use as their source code a sentence or a paragraph. Another example we will see when discussing performance is a scanner and parser that uses XML as its source code (XML is not "code" in the programming sense of the word). The example source code we use throughout most of this paper is an ML program so it is in fact a source code in the more well-known definition of the term.

Tokens

A token is a sequence of one or more characters. Given the English language, "a" is a token consisting of one character and "an" and "at" are tokens consisting of two characters. A token can be defined as a rule in such a way that many different character sequences satisfy the rule. For example, in Java, a variable name token is a sequence of characters that begins with a letter followed by zero or more other letters or numbers. In that case, "a", "a100" and "aaa" are all three valid variable name tokens. However, "100a" is

not a valid variable name token. As you can see, there are an infinite number of valid sequences of characters that are valid variable name tokens.

Sometimes you want a token to be much more precise so there are not an infinite number of character sequences that match it. For example, you can have an end token that starts with the "e" symbol followed by the "n" symbol and ends with the "d" symbol. Hence the token end is only satisfied by the following sequence of characters: "end".

You may have noticed that the sequence of characters "end" would actually satisfy the rule to be an end token and also to be a variable name token. So which is it? It depends. The scanner will be responsible for making that determination. Generally speaking, scanners assign a sequence of symbols to the token that has the strictest rule. So "end" would be assigned to be an end token and not a variable name token which would prevent you from using "end" anywhere the language expects a variable name. Indeed, that is the case in Java where "end" is a reserved word.

The spelling of a token is not always meaningful. In the English language, the spelling of a word is important because you need to know the spelling to know the definition. If you are told that a sentence consists of a subject token followed by a verb token followed by a noun token followed by a period token you would not know the meaning of the sentence without knowing the spelling of the first 3 tokens. Once you know that the tokens are spelled "I" "see" "things" "." you can understand the meaning. However, the spelling of the period token "." does not add any new meaning to the sentence. If we decide to spell periods "*" for example, the sentence would not change its meaning. The presence of a period token as opposed to say a question mark token is what is important, not the spelling of that token itself.

Regular Expressions

A regular expression, usually abbreviated to regex to regexp, is a "convenient notation for expressing a set of strings of terminal symbols"[ii]. A regex itself is a string that contains special characters that have special meaning. Often times if you want to use the special character without the special meaning associated with it you have to escape out the special character. Regular expressions are generally used for pattern searching or filtering. Typically you will write a regular expression and look for a string that matches the regex. Regexes are best understood by looking at examples.

- The regex "josh" expresses the set containing only one string { "josh" }. Only the string "josh" matches this regex. Although other strings such as "josh foure" and "my name is josh" contain the substring "josh", they are not a match to the regex. The regex "josh" introduces the concept of concatenation because "josh" is really interpreted as "j" followed by "o" followed by "s" followed by "h".

- The regex "jo|sh" introduces the special character "|" that means "or". This regex generates a set containing 2 strings { "jo", "sh"}. Note that concatenation takes precedence over alternatives. If this were not the case, the regex would still generate 2 strings but they would be "joh" and "jsh". The next rule will show what would happen if alternatives happened before concatenation.

- The regex "j(o|s)h" introduces the concept of grouping by using parentheses. This generates the set of 2 strings { "joh", "jsh" }.

- The regex "jo(sh)*" introduces the concept of iteration using the special character "*". The "*" means zero or more times. Hence, the set generated by this regex is infinite { "jo", "josh", "joshsh", "joshshsh", … }.

- The regex "jos\d" introduces the special character "\d" which means any digit. Hence the regex corresponds to a set of 10 strings because the "\d" can be replaced by any one of 10 digits 0-10. Therefore the matching set of strings is { "jos0", "jos1", …, "jos9" }.

- The regex "jos[eht]" introduces the special characters "[" and "]" which mean one of the characters in the brackets. The regex therefore corresponds to a set containing 3 entries { "jose", "josh", "jost" }.

Unfortunately there are different specifications of regular expressions. Much like the old days (and to some extent still today) in the browser wars when Microsoft and Netscape had different extensions to javascript, so it is with different regular expression engines that have added special symbols that they have decided their customers need. If you use the basic regular expression symbols you can be almost assured of their portability but I wanted to point out that not every expression is portable. Furthermore, although Perl5 is the most used regex engine out available, we will be using the rules from the Java 1.4 JDK since this thesis was written in Java. Java did not actually support regular expressions in the core JDK until version 1.4 so you will need that version or higher to run the Parser Generator.

Below you will find a table containing a recap of the examples above as well as some additional useful regular expression special characters. See the Sun documentation[5] for more details.

Regex	Description
X	The character x.
XY	The concatenation of the strings generated by X and by Y.
X|Y	Any string generated by either X or Y.
(XY)	The grouping of XY. See above.
X*	The concatenation zero or more times of the string generated by X.
X?	The string generated by X once or not at all.
X+	The concatenation of the string generated by X one or more times.
\xhh	The character whose hex value is hh. For example, a double quote would be \x22.
[aeiou]	One of the characters in the brackets. In this case either 'a', 'e', 'i', 'o' or 'u'.
[a-d]	One of the character in between 'a' and 'd' inclusive. In this case 'a', 'b', 'c' or 'd'. Same as [abcd].
\d	A digit. Same as [0-9] or [0123456789].
\w	A word character [a-zA-Z_0-9].
^x	Any character except x.

Table 1: Useful regular expression.

Regular expressions are extremely powerful when creating complicated string patterns. Their main drawback is that they tend to be easier to write than to read and can therefore be difficult to maintain. Another problem with regular expressions is that they do not exhibit self-embedding. You can not define a regex in terms of other regexes without copying it and pasting it. Because of this latter problem, we will not use regular expressions when defining a grammar. Some of the concepts of regular expressions are, however, embedded in EBNF (the first 6 in Table 1) as we will see shortly and we will use regular expressions when creating a scanner.

There are many regular expression resources on the internet. A quick search of Google will return many tutorials including the one of Jan Goyvaerts[6], which is especially helpful.

Introduction to ML

In this paper, we will use a subset of ML as the language we want to generate a scanner and parser for. In fact, the subset of ML we will use will be a little more strict with type definition (for example, "int" will be a reserved word so you can not redefine it) so every program in our subset language will be an ML program but not vice versa. Unlike Java, which is an object oriented language, ML is a functional language. Instead of traditional variables and methods, ML programs are defined as a series of mathematical functions that are then executed or called. Looping in ML is done via recursion rather than procedure "for" loops. One of the driving forces behind using functional languages is that "functional languages such as ML, Hope and Lisp allow us to develop programs which will submit logical analysis relatively easily. Using a functional language we can make assertions about programs and prove these assertions to be correct."[iii]

You do not actually need to know anything more about ML to follow the discussion on how to parse ML source code, however there are some good tutorials online [8]. You can also download an interpreter [7] to run the programs although that is not necessary either. Below is a sample program:

```
fun fact(n:int):int = if n=0 then 1 else n*fact(n-1);
fact(4);
```

This program defines one function named "fact" that take a single integer as input and returns an integer as output. "fact" is defined as a conditional expression that returns the factorial of the input using recursion. On the second line the function "fact" is called with an input of "4". The result of

this call could be 4! which is 24. This is the source program that we will be using throughout this paper.

Here is a more complicated example that shows two function declarations:

```
fun dec(a:int):int = a-1;
fun pow(x:int, y:int):int = if y=0 then 1 else x*pow(x,dec(y));
pow(2,3);
```

This program calculates x to the power of y, or in this case 2^3 which is equal to 8. The "dec" function is not actually necessary, since "dec(y)" could have been replaced with "y-1" in the "pow" declaration, but I this is a good demonstration of two functions working together.

Scanning

This paper focuses primarily on parsers so we will only briefly discuss how scanners work. To test a parser you need a scanner, so I had to generate a scanner for each parser that I generated.

The ML-subset language uses 34 tokens. 31 of the tokens are exact string matches: "fun", "int", "real", "string", "list", "else", "if", "then", ":", ";", ",", "~", "not", "(", ")", "[", "]", "and", "*", "/", "mod", "div", "=", ">", ">=", "\=", "<", "<=", "-", "or" and "+". The remaining 3 tokens are IntegerLiteral, StringLiteral and Identifier. An IntegerLiteral token is a digit (0 through 9) followed by zero or more digits. A StringLiteral is a double quote followed by any number of characters that aren't double quotes and that ends with a double quote. An Identifier is a letter character ('a' through 'z' lower case and 'A' through 'Z' upper case) followed by zero or more letters or digits. Clearly a computer would not understand this English explanation. We will define these tokens using regular expressions in the Thesis Implementation section.

As the scanner reads in each character from the source program, it must determine which token the character belongs to. The following listing shows the output of the scanner for the sample ML-subset program. Each line gives the following pieces of information: the line number, the string that the scanner identified as a token, the kind of token and the token variable name that corresponds to the token kind.

```
Line: 1, spelling = [fun], kind = 3, Token.FUN
Line: 1, spelling = [fact], kind = 0, Token.IDENTIFIER
Line: 1, spelling = [(], kind = 16, Token.LPAREN
Line: 1, spelling = [n], kind = 0, Token.IDENTIFIER
```

```
Line: 1, spelling = [:], kind = 11, Token.COLON
Line: 1, spelling = [int], kind = 4, Token.INT
Line: 1, spelling = [)], kind = 17, Token.RPAREN
Line: 1, spelling = [:], kind = 11, Token.COLON
Line: 1, spelling = [int], kind = 4, Token.INT
Line: 1, spelling = [=], kind = 25, Token.EQUAL
Line: 1, spelling = [if], kind = 9, Token.IF
Line: 1, spelling = [n], kind = 0, Token.IDENTIFIER
Line: 1, spelling = [=], kind = 25, Token.EQUAL
Line: 1, spelling = [0], kind = 1, Token.INTEGERLITERAL
Line: 1, spelling = [then], kind = 10, Token.THEN
Line: 1, spelling = [1], kind = 1, Token.INTEGERLITERAL
Line: 1, spelling = [else], kind = 8, Token.ELSE
Line: 1, spelling = [n], kind = 0, Token.IDENTIFIER
Line: 1, spelling = [*], kind = 21, Token.ASTERISK
Line: 1, spelling = [f], kind = 0, Token.IDENTIFIER
Line: 1, spelling = [(], kind = 16, Token.LPAREN
Line: 1, spelling = [n], kind = 0, Token.IDENTIFIER
Line: 1, spelling = [-], kind = 31, Token.MINUS
Line: 1, spelling = [1], kind = 1, Token.INTEGERLITERAL
Line: 1, spelling = [)], kind = 17, Token.RPAREN
Line: 1, spelling = [;], kind = 12, Token.SEMICOLON
Line: 2, spelling = [f], kind = 0, Token.IDENTIFIER
Line: 2, spelling = [(], kind = 16, Token.LPAREN
Line: 2, spelling = [4], kind = 1, Token.INTEGERLITERAL
Line: 2, spelling = [)], kind = 17, Token.RPAREN
Line: 2, spelling = [;], kind = 12, Token.SEMICOLON
Line: 2, spelling = [], kind = -1, Token.EOT
```

Figure 1: Output of ML-subset scanner.

The scanner completes successfully for our sample source program so we are assured that each character in the program is allowed and each sequence of characters is also valid. However, the scanner does not guarantee that a "fun" token followed by an Identifier token is a valid program. That is the responsibility of the parser.

Syntactical Grammars

The lexical grammar that a scanner uses establishes which tokens are valid for a language. A syntactical grammar then defines the order in which the tokens can be arranged. Even though the token "the" is valid in the English language, the string "the the the" makes no sense.

Context-free Syntax

When trying to communicate with a computer, you must use a rigorous formal specification of the language you will be using. A context-free grammar consists of the following 4 elements:

1. A finite set of terminal symbols (or just terminals) that are either one or more characters concatenated together. For example, in the ML-subset language "fun", "(" and ">=" are three different terminals. A terminal is said to be atomic in that it is represented by a single token.

2. A finite set of nonterminal symbols which represent a grouping of terminals and nonterminals. In ML-subset, *Program*, *Functions* and *ParList* are nonterminals.

3. A start symbol which is one of the nonterminals. The Parser Generator requires that the start symbol must be *Program* for each EBNF because of how it was coded however that is not a normal limitation on grammars. You will notice that the ML-subset EBNF's first production rule will define *Program*.

4. A finite set of production rules that define how the nonterminals, and therefore the grammar, are composed.

BNF Grammar

The rules for the context-free grammar listed above are usually written in BNF (Backus-Naur Form) notation. "In BNF, a production rule is written in the form N ::= α, where N is a nonterminal, and where α is a (possibly empty) string of terminal and/or nonterminal symbols."[iv] For example, a simple grammar in BNF notation is:

```
Program     ::= "Hello" Place.
Place       ::= "World".
```

This consists of a language with 2 nonterminals Program and Place and 2 terminals "Hello" and "World". The start symbol is *Program* as expected. In theory you can have several production rules for the same nonterminal such as:

```
Place       ::= "World".
Place       ::= "Planet".
Place       ::= "Earth".
```

However those are usually (and must be for the Parser Generator) grouped as a single rule with a "|" being used in between each definition:

```
Place ::= "World" | "Planet" | "Earth".
```

You will recall that the "|" is used in regular expressions to indicate alternatives as well.

EBNF Grammar

Extended BNF or EBNF is an extension of BNF where the production rule definitions cannot just contain a sequence of terminal and nonterminals or alternatives of sequences, but can also contain some regular expression syntaxes. Specifically, EBNF introduces 3 aspects of regular expressions:

1. In EBNF, you can use the concept of grouping in the production rule definitions by surrounding the symbols you want to group in parenthesis "(" and ")". This is the same rule as for regular expressions.

2. In EBNF, you can use the concept of zero or more grouping. In regular expression notation "*" is used which is also sometimes used in EBNF notation as well. However we will adopt the notation of surrounding the symbols that can appear zero or more times in curly brackets "{" and "}". Thus if we want to say that the nonterminal A can appear zero or more times we would express it as { A }.

3. In EBNF, you can use the concept of zero or one grouping. In regular expression notation "+" is used, however we will adopt the notation of surrounding the symbols that can appear zero or one time in square brackets "[" and "]". Thus if we want to say that the nonterminal A can appear zero or one time we would express it as [A]. Note that the square brackets are used in regular expressions to define a set of possible results but that they are not needed in EBNF because the alternative "|" does the same thing.

Using EBNF simplifies expressing certain rules compared with using BNF. However, at the end of the day you cannot express more in EBNF than in BNF. Since EBNF adds optional regular expression constructs, every BNF is actually an

EBNF but not vice versa. We will now look at 3 grammar simplifications you can make when going from BNF to EBNF:

1. Substitution of nonterminals: Using groupings (#1 above) can help you simplify a grammar by not having to define certain simple nonterminals. For example, the following grammar that is 2 lines in BNF is only 1 line in EBNF:

   ```
   BNF:        A ::= "A" B | "C".
               B ::= "B1" | "B2".

   EBNF:       A ::= "A" ("B1" | "B2") | "C".
   ```

   Note you could substitute nonterminals in BNF if and only if they were not recursive sequences. For example,

   ```
   BNF:        A ::= "A" B | "C".
               B ::= "B1" "B2".

   BNF:        A ::= "A" "B1" "B2" | "C".
   ```

   However, the addition of the grouping lets you substitute more complicated nonterminals including ones that are defined with alternatives.

2. Left factorization: Using groupings (#1 above) you can factorize a production rule definition that contains alternatives just like you would in multiplication:

   ```
   BNF:        A ::= "B" "C" | "B" "D".

   EBNF:       A ::= "B" ("C" | "D").
   ```

   If your grammar contains 2 alternatives where one is a subset of the other, you can use left factorization and zero or one grouping (#3 above) as follows:

```
BNF:        A ::= "B" | "B" "C".

EBNF:       A ::= "B" ["C"].
```

3. Elimination of left recursion: Using the zero or more grouping (#2 above) notation of EBNF, you can express recursion more concisely than in BNF:

```
BNF:        A ::= "B" | A "C".

EBNF:       A ::= "B" { "C" }.
```

   It is much easier to see what is going on, namely that A starts with "B" and is followed by zero or more "C", using the EBNF notation than the BNF notation.

Now that we understand how to interpret the EBNF notation, let us look at the EBNF of a real language, ML-subset:

```
Program ::= { Functions ";" } Call ";".
Functions ::= "fun" Identifier "(" [ParList] ")" ":" Type "=" Expression.
ParList ::= Par { "," Par }.
Par ::= Identifier ":" Type.
Type ::= SimpleType ["list"].
SimpleType ::= "int" | "real" | "string".
Expression ::= ConditionalExpression | BasicExpression.
ConditionalExpression ::= "if" BasicExpression "then" Expression "else"
Expression.
BasicExpression ::= PrimaryExpression {PrimaryOperator PrimaryExpression}.
PrimaryOperator ::= "or" | "and".
PrimaryExpression ::= SimpleExpression [RelationalOperator
SimpleExpression].
RelationalOperator ::= "<" | "<=" | ">" | ">=" | "=" | "\=".
SimpleExpression ::= ["~"] Term {AddingOperator Term}.
AddingOperator ::= "+" | "-".
Term ::= Factor {MultiplyingOperator Factor}.
MultiplyingOperator ::= "*" | "/" | "div" | "mod".
Factor ::= IntegerLiteral | StringLiteral | Identifier ["(" [ArgList] ")"]
| ListLiteral | "(" Expression ")" | "not" Factor.
ListLiteral ::= "[" [ExpressionList] "]".
ExpressionList ::= Expression { "," Expression }.
Call::= Identifier "(" [ArgList] ")".
ArgList ::= Expression {"," Expression }.
```

Figure 2: EBNF of ML-subset

As you can see, this ML-subset contains 21 production rules. Without knowing anything about ML-subset you can learn a lot just by looking at the EBNF. You can see that each ML-subset program starts with zero or more optional function declarations followed by a function call and terminating with a semi colon. Each function begins with the reserved word "fun" and ends with a semi colon. The function call begins with an *Identifier* followed by zero or one *ArgList* in parenthesis. We will not go through each rule, but this should give you an idea.

If you start with the *Program* nonterminal and write out its definition and replace each nonterminal in the definition with its definition and so forth until you are only left with terminals you will end up with a program that parses correctly. The program may violate other rules set forth in the ML-subset specification but it will succeed in parsing correctly. For example, "call josh();" is a valid program that will parse but it would not execute because the function "josh" is not defined.

Starter Sets

Given a nonterminal A, it is often very useful to know what terminals can start that production rule. The notation is as follows: starters("a" "b" | "c") = { "a", "b"}. There are 5 rules:

1. starters(ε) = { }
2. starters("a") = { "a" }
3. starters(A B) = starters(A) ∪ starters(B) if A can be empty
   = starters(A) if A can not be empty
4. starters(A | B) = starters(A) ∪ starters(B)
5. starters({A}) = starters(A)

When determining the starter sets it is typically easier to start at the last production rule and work your way up. The reason for this is that nonterminals at the end of the grammar tend to be defined of more terminals than the ones at the top. The top nonterminals tend to define the structure of the grammar whereas the bottom ones have the specific terminals. When calculating the starter sets for a nonterminal, if the definition starts with a nonterminal, look at your table to see if you have already found the starter set of that nonterminal. If you have, simply add that set to the starter set of the current nonterminal. If you have not already found the starter set, add the nonterminal in the starter set column for the current nonterminal and replace it as soon as you have found the starter set for that nonterminal.

Nonterminal	**Starter Set**
Program	"fun", Identifier
Functions	"fun"
ParList	Identifier
Par	Identifier
Type	"real", "string", "int"
SimpleType	"real", "string", "int"
Expression	IntegerLiteral, "(", "not", StringLiteral, "[", Identifier, "if", "~"
ConditionalExpression	"if"
BasicExpression	IntegerLiteral, "(", "not", StringLiteral, "[", Identifier, "~"
PrimaryOperator	"and", "or"
PrimaryExpression	IntegerLiteral, "(", "not", StringLiteral, "[", Identifier, "~"
RelationalOperator	"<=", ">", "\=", ">=", "="
SimpleExpression	IntegerLiteral, "(", "not", StringLiteral, "[", Identifier, "~"
AddingOperator	"-", "+"
Term	IntegerLiteral, "not", "(", "[", StringLiteral, Identifier
MultiplyingOperator	"mod", "/", "*", "div"
Factor	IntegerLiteral, "not", "(", "[", StringLiteral, Identifier
ListLiteral	"["
ExpressionList	IntegerLiteral, "not", "(", "[", StringLiteral, Identifier, "if", "~"
Call	Identifier
ArgList	IntegerLiteral, "not", "(", "[",

	StringLiteral, Identifier, "if", "~"

Table 2: Starter sets for EBNF of ML-subset

Follow Set

Given a nonterminal A, it is often very useful to know what terminals can be found after that production rule. Unlike with the starter sets, figuring out the follow set of each nonterminal is not sufficient. The follow sets of the nonterminals are used to help determine the follow sets for the production rules that can be empty.

NonTerminal	**Follow Set**
Program	EOT
Functions	";"
ParList	")"
Par	",", ")"
Type	"=", ",", ")"
SimpleType	"list", "=", ",", ")"
Expression	";", "else", ",", ")", "]"
ConditionalExpression	";", "else", ",", ")", "]"
BasicExpression	"then", ";", "else", ",", ")", "]"
PrimaryOperator	IntegerLiteral, "(", "not", StringLiteral, "[", Identifier, "~"
PrimaryExpression	"or", "and", "then", ";", "else", ",", ")", "]"
RelationalOperator	IntegerLiteral, "(", "not", StringLiteral, "[", Identifier, "~"
SimpleExpression	"<", "<=", ">", ">=", "=", "\=", "or", "and", "then", ";", "else", ",", ")", "]"
AddingOperator	IntegerLiteral, "not", "(", "[", StringLiteral, Identifier
Term	"+", "-", "<", "<=", ">", ">=", "=", "\=", "or", "and", "then", ";", "else", ",", ")", "]"
MultiplyingOperator	IntegerLiteral, "not", "(", "[", StringLiteral, Identifier
Factor	"*", "/", "div", "mod", "+", "-", "<", "<=", ">", ">=", "=", "\=", "or", "and", "then", ";", "else", ",", ")", "]"
ListLiteral	"*", "/", "div", "mod", "+", "-", "<", "<=", ">", ">=", "=", "\=", "or", "and", "then", ";", "else", ",", ")",

	"]"
ExpressionList	"]"
Call	";"
ArgList	")"

Table 3: Follow sets for EBNF of ML-subset

LL(1) Grammar

Some grammars have certain types of properties that make them easire to work with such as being LL(1). A grammar is said to be LL(1) if it satisfies the following 2 rules:

1. If the grammar contains a production rule with alternatives, then the starter sets of each alternative must be disjoint. Put another way, if we have A::= A1 | A2 | … | AN then starters(Ai) and starters(Aj) are disjount for all i and j where i≠j.

2. If a production rule A or part of the production rule N can be empty, then the starter set of N must be disjoint from the follow set of the grouping in that context. For example, if you have A::= {A1} A2 then starters({A1})=starters(A1) must be disjoint from follow({A1}).

We will now show that ML-subset satisfies rule #1 by looking at the nonterminals defined by alternatives. There are 7 such rules:

Nonterminal	**Alternative**	**Alternative Starter Set**
SimpleType	"int"	"int"
	"real"	"real"
	"string"	"string"
Expression	ConditionalExpression	"if"
	BasicExpression	IntegerLiteral, "(", "not", StringLiteral, "[", Identifier, "~"
PrimaryOperator	"or"	"or"
	"and"	"and"
RelationalOperator	"<"	"<"

	"<="	"<="
	">"	">"
	">="	">="
	"="	"="
	"\="	"\="
AddingOperator	"+"	"+"
	"-"	"-"
MultiplyingOperator	"*"	"*"
	"/"	"/"
	"div"	"div"
	"mod"	"mod"
Factor	IntegerLiteral	IntegerLiteral
	StringLiteral	StringLiteral
	Identifier["(" [ArgList] ")"]	Identifier
	ListLiteral	"["
	"(" Expression ")"	"("
	"not" Factor	"not"

Table 4: Showing that ML-subset satisfies rule #1 of being an LL(1) grammar.

The table demonstrates that the starter set of each alternative is disjoint from the starter set of each other alternative for each nonterminal.

We will now show that ML-subset satisfies rule #2 by showing the starter set of each group that can be empty is disjoint from its follow set. When generating this table, it is often useful to refer back to the tables showing the starter sets and follows sets of the EBNF. If a grouping that can be empty is before another symbol, then its follow set is the starter set of that symbol. If the grouping that can be empty is at the end of a production rule, then its follow set is the same as the follow set of its nonterminal.

Groupings that can be empty	**Starter set of grouping**	**Follow set of grouping**
{ Functions ";" }	"fun"	Identifier
[ParList]	Identifier	")"
{ "," Par }	","	")"
{PrimaryOperator PrimaryExpression}	"and", "or"	"then", ";", "else", ",", ")", "]"
[RelationalOperator SimpleExpression]	"<=", ">", "\=", ">=", "="	"or", "and", "then", ";", "else", ",", ")", "]"

["~"]	"~"	IntegerLiteral, "not", "(", "[", StringLiteral, Identifier
{AddingOperator Term}	"-", "+"	"<", "<=", ">", ">=", "=", "\=", "or", "and", "then", ";", "else", ",", ")", "]"
{MultiplyingOperator Factor}	"mod", "/", "*", "div"	"+", "-", "<", "<=", ">", ">=", "=", "\=", "or", "and", "then", ";", "else", ",", ")", "]"
[ArgList]	IntegerLiteral, "not", "(", "[", StringLiteral, Identifier, "if", "~"	")"
[ExpressionList]	IntegerLiteral, "not", "(", "[", StringLiteral, Identifier, "if", "~"	"]"
{"," Expression}	","	"]"
[ArgList]	IntegerLiteral, "not", "(", "[", StringLiteral, Identifier, "if", "~"	")"
{"," Expression}	","	")"

Table 5: Showing that ML-subset satisfies rule #2 of being an LL(1) grammar.

The table demonstrates that the starter set of each grouping that can be empty is disjoint from its follow set.

Grammar Transformations to make a Grammar LL(1)

The input to the Parser Generator needs to be an LL(1) grammar in EBNF format. There are 3 transformations that can be applied to a non-LL(1) grammar to make them LL(1). We discussed the first 2 rules when looking at how EBNF syntax can help us simplify grammars previously expressed in BNF. However, we now look at the same rule from a starter set and LL(1) perspective to show when we have to apply these rules. We will see these rules and examine why they are necessary:

1. Left Factorization:

Suppose you have a production rule A::= B C | B D. This clearly violates the first rule of LL(1) grammars because starters(B C) and starters(B D) are not disjoint. In fact, if B can not generate the empty set then starters(B C) is equal to starters(B D) which is equal to starters(B). The solution is to left factorize B so you end up with A::= B (C | D).

2. Left Recursion Elimination:

   Suppose you have a production A::= X | A Y. Once again this violates the first rule of LL(1) grammars. Indeed, starters(X) is equal to starters(A Y) since starters(A Y)=starters(X).
   The solution is to replace the rule A::= X | A Y with A::={X} Y. Note that if starters(X) and starters(Y) are not disjoint then this will violate rule 2 of LL(1) grammars.

3. Associative Property of Groups
   Suppose you have a production rule A::={X} X. This will violate the second rule of LL(1) grammars because starters({X}) = follow({X}) = starters(X).
   The solution is to reverse the one or many grouping with the non terminal. Change A::={X} X to A::=X {X}. You still need to make sure that starters({X}) is disjoint from follow({X}) which is equal to follow(A) to make sure the language is LL(1). The same rule applies to A::=[X] X being replaced to A::=X [X].

The Parser Generator does not perform these transformations. It is the responsibility of the user to make the transformations necessary to ensure that the input EBNF is LL(1).

Abstract Syntax Trees (AST)

The EBNF of a language specifies precisely in what order tokens must be found. However, many of the terminal tokens are present only to distinguish one nonterminal from another and do not have any bearing on the semantics of the program. For example, in ML-subset the "fun" token is needed to identify function declarations but the token "fun" does not actually help in the function declaration itself. The Abstract Syntax Tree, or AST, lets you see the phrase structure of a language without irrelevant terminal tokens. For example, the first few lines of the ML-subset AST grammar look like this (we will see the entire tree later):

```
Program ::= ProgramFunctionsZeroOrMore Call.
ProgramFunctionsZeroOrMore ::= Functions ProgramFunctionsZeroOrMore .
Functions ::= Identifier FunctionsParListZeroOrOne Type Expression .
etc.
```

This gives us a lot of information about the phrase structure of an ML-subset program. It is clear from this simplified grammar that a *Program* consists of zero or more *ProgramFunctions* followed by a single Call. The *ProgramFunctions* in turn define a *Functions* which can be repeatable. Each *Functions* consists of an *Identifier*, a *FunctionsParListZeroOrOne*, a *Type* and an *Expression*.

```
EBNF:  Functions ::= "fun" Identifier "(" FunctionsParListZeroOrOne ")"
":" Type "=" Expression .

AST: Functions ::= Identifier FunctionsParListZeroOrOne Type Expression .
```

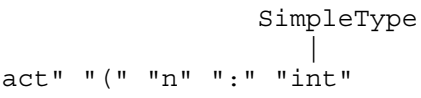
If you compare the EBNF production rule for Functions and the AST rule you will see all that all the key elements are the same: *Identifier*, *FunctionsParListZeroOrOne*, *Type* and *Expression*.

Parsing

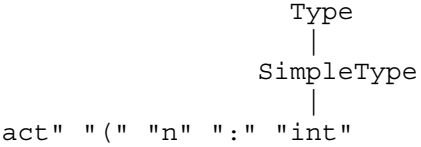
Parsing is the act of taking a sequence of tokens and making sure that they constitute a valid sentence in the grammar. Conceptually parsing is similar to scanning except that you work with tokens instead of characters. You build a sentence by following a production rule as opposed to building a token by following some other sort of rule such as a regular expression. There are 2 main techniques in parsing: bottom-up parsing and top-down parsing.

1. **Bottom-up parsing**: The parser reads the tokens from left to right and tries to find a match for each production rule. Here are the first 3 steps in the bottom-up parsing of our ML-subset example:

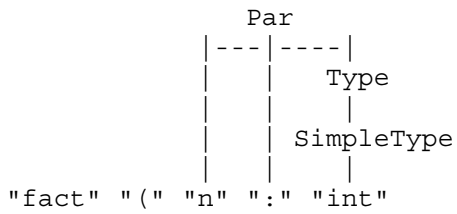
   a. The parser reads the "fun" token but that is not a match to any production rules so it reads "fact", "(", "n", ":", "int". When the "int" token is reached, it finds a match and identifies that "int" is a SimpleType.



   b. The next token ")" is now read. This means that the SimpleType "int" is a Type.



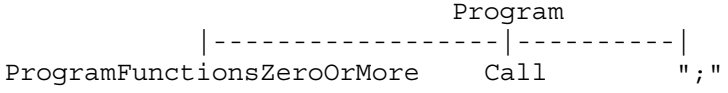
c. The parse now recognizes that it has an Identifier "n" token, a terminal ":" token followed by a Type. This is Par.



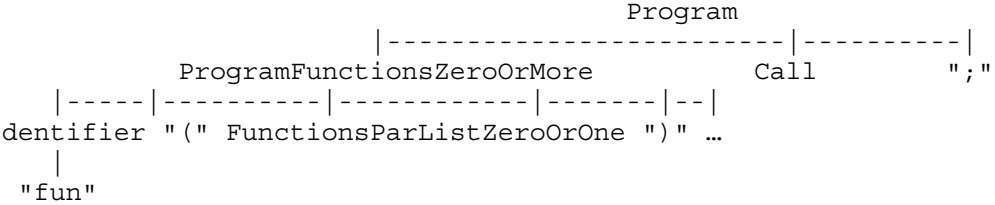
This technique continues until we have an entire AST tree built from the bottom up.

2. **Top-down parsing**:

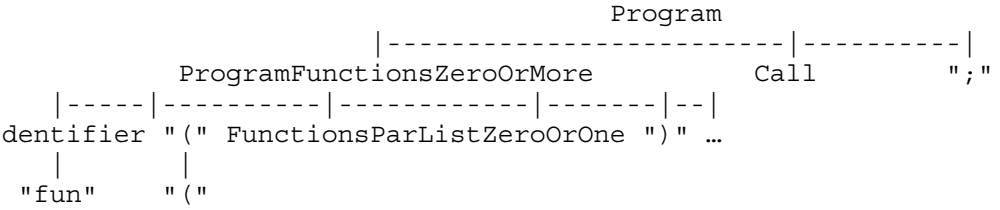
a. The top node in the tree is always Program so that node is created. 3 stubs are created for ProgramFunctionsZeroOrMore Call ";".



b. The first token "fun" is read in. This matches the first terminal in the Functions production rule so a Functions object is created with a stub for the other elements in function that it expects: Identifier "(" FunctionsParListZeroOrOne ")" ":" Type "=" Expression.



c. The next token "(" is read in and this is what we expected so we connect that to the stub.



This technique continues until we have an entire AST tree built from the top down.

Parser Generator Overview

The purpose of this project is to allow a user to define a grammar and a lexicon and to automatically generate the scanner and parser. This is a task that is very suitable for a program because an algorithm can faithfully be followed to generate all the files needed to parse a grammar.

Something to keep in mind while reading is that the Parser Generator itself contains a scanner and parser and it reads in an EBNF grammar. In memory, the EBNF grammar is represented by EBNF AST objects. The EBNF grammar is then transformed into an AST grammar which is represented in memory by AST AST objects. Finally the output of the program is a new scanner and parser and set of AST objects that will represent a source program of the initial EBNF grammar.

Inputs

- Grammar Input: The user enters the grammar in EBNF format. We have seen the EBNF of ML-subset and we will see the EBNF of an XML-subset in the "Performance" section. Additional EBNFs are included in appendixes F1-F6.

- Token Input: The token definition is entered in a proprietary format that we will see in the next section. For each token, a token variable name that satisfies Java variable name standards must be given along with the regular expression that defines a match.

- Sample Code Input: The sample code is not actually used by the parser generating code per se. However, it is outputted back out with the new generated files so that they can easily be tested.

Outputs

The Parser Generator creates 64 classes for the ML-subset language. Without going into too many implementation details, the following is a high level overview of the major output classes.

1. Token.java: This class contains information about each Token. Each token has a unique id and a unique name that is defined in this file because programs work better with ids than with names.
2. Scanner.java: This class takes a source program (or technically any InputStream) and breaks it up into tokens. Each token is an instance of the Token class that contains its type and spelling.
3. Parser.java: This class uses the scanner to get a stream of tokens and generates an AST tree representing the source program. The parser that the Parser Generator generates is a top-down parser.
4. Visitor.java: This interface allows other classes to be written than will easily traverse the AST tree.
5. AST classes: Each class represents a grammar production rule. The classes don't necessarily map exactly to the user-entered EBNF grammar because that grammar is transformed into a more computer-friendly AST grammar.

The generated code can be found in Appendix J.

Parser Generator Implementation

A language is a combination of 3 things: a set of rules that defines which characters are allowed and in what order, a set of rules that define how the tokens can be ordered, and the meaning behind the tokens. A scanner program is responsible for reading the characters in a source code file and making sure they are allowed and in an acceptable order to form a token. A parser program then looks at each token to make sure that it is in a correct order. A checker program can then check that the correctly ordered tokens have some meaning. The "meaning" behind the source code may be interpreted countless ways depending on the language. There are no standard set of rules you can give a computer for it to understand meaning. The Parser Generator therefore focuses on generating the scanner and the parser for which you can express rules in a computer-readable format.

Let us revisit the ML-subset example and see how a scanner and a parser would interact. You will recall the sample ML-subset source code as:

```
fun fact(n:int):int = if n=0 then 1 else n*fact(n-1);
fact(4);
```

The scanner would read each character and identify which sequence of characters forms which token. Figure 1 shows that the scanner would identify 32 tokens starting with "fun" then "fact" then "(" and so forth. The parser will retrieve each token and make sure that the tokens are ordered correctly while generating an in-memory AST tree that will be returned by the parser. The AST tree contains all the information of the ML-subset source program. In this case the parser would see that the first token is a "fun" token so it would know that the source code begins with a function declaration so the following token must be an identifier, which "fact" is, followed by an open parenthesis token, which is

also the case. A checker would then take the AST tree generated by the parser and go through it and do the appropriate validation. In the ML-subset case, it would check that the method being called is defined in a function declaration. In this case "fact" is indeed defined prior to being called. A checker would also make sure that all the variables being used are defined. In this case there is only one variable "n" which is first defined and then used 3 times. Other checks would include type checking, for example making sure that you don't define a function with an "int" parameter and then pass in a "string" when it is called. The following source code is syntactically a correct ML program but it has no meaning and it will not run because it would violate all three checks defined above:

```
fun factorial(a:string):int = if b=0 then 1 else n*factorial(c-1);
fact(4);
```

Indeed, the method "fact" being called is not defined. Neither are the variables b and c. Lastly, the call to "factorize" expects a string but clearly "c-1" is an int. In some other languages, such as Javascript (but not Java), variables do not need to be defined before they are assigned. If this were the case in ML, then "b=0" would not be an error. This just goes to show that a similar construct can be valid in one language and invalid in another.

Scanner Implementation

A typical scanner code consists of 2 classes called Token and Scanner (in files called Token.java and Scanner.java). The Scanner is responsible for reading an InputStream and converting sequences of characters into the appropriate Token objects. The scanner takes a BufferedReader in its constructor and provides a public "scan()" method that returns a Token object. The Token class is nothing more than a data class in which each instance contains the token type, the spelling of the token and the line number in which the token was found. For each token type, there is a public final static constant identifier in the Token class that the parser or any other class can use to refer to the token type without referring to the type number directly. For example, "public final static byte Token.FUN=3;" is defined in the ML-subset Token. This allows a programmer to check that the token corresponding to "fun" is equal to Token.FUN instead of just the number 3 which makes the code easier to understand. Here are outlines for both the Token and the Scanner:

Token:

```
public class Token {

      public int kind;
      public String spelling;
      public int line;

      public Token(int kind, String spelling, int line){
      this.kind = kind;
      this.spelling = spelling;
      this.line = line;
      }

      public final static byte
            IDENTIFIER = 0
            INTEGERLITERAL = 1,
            STRINGLITERAL = 2,
            FUN=3,
            …  //other constants
            PLUS = 33;
```

```
            //getter methods for king, spelling and line
      }
```

Scanner:

```
      public class Scanner{
            private char currentChar;
            private byte currentKind;
            private BufferedReader inFile;
            … //other variables

            public Scanner(BufferedReader inFile){
                  this.inFile = inFile;
                  … //other constructor code
            }

            …  //more scan methods like scanSeperator and scanToken

            public Token scan(){
                  currentSpelling = new StringBuffer("");
                  while(currentChar == '!' || currentChar == ' ' || currentChar
            == '\n' || currentChar == '\r')
                        scanSeparator();
                  currentKind = scanToken();
                  return new Token(currentKind, currentSpelling.toString(),
            line);
            }

      }
```

Different scanners implement the "scan()" method differently. If we do not like the implementation or want to optimize it, we are free to change it as long as it returns the same tokens that the other implementation returned. In fact, any scanner implementation for the same language should always return the same set of tokens. If one scanner returned one set and another returned a different set for the same exact source program, we would have an ambiguous language.

In order to auto-generate a scanner, we need to determine how the user will enter the token rules. In the "scanning" section earlier we explained the rules in English, which is fine for human readers of this document but not acceptable as an input to the Parser Generator. Unfortunately, there is not a

set standard on how to define token rules unlike the EBNF standard that exists to express grammars. We will first look at David A Watts[1]' approach which also uses EBNF to express token rules but that has some limitations. We will then look at the approach I developed.

The problem with the approach that David A Watts[1] takes, is that some tokens rules are specified in the EBNF of the grammar as terminals while others are defined in the lexicon-specific rules. For example in ML-subset, "real" and "string" are defined in the EBNF of the grammar but *Identifier* is specified in the lexicon. Watts suggests that all terminals specified in the EBNF are simply a special case of an Identifier. Thus, in order to systematically generate a scanner, he proposes checking each of the Identifiers against all of the terminals defined in the grammar EBNF and if there is a match returning a special token that corresponds to the terminal. For example, in the ML-subset example, the scanner would first read in the string "fun" and determine that it was an Identifier token. It would then check to see if this identifier string matched one of the terminals defined in the grammar EBNF. If that were the case, it would return the token specific to the terminal. If no match was found, it would return the identifier token.

Checking Identifiers to see if they are equal to the terminals works well when the terminals are reserved words in a language. For example, in ML-subset we have the terminals "real" and "string" and we do have a nonterminal *Identifier*. In this case, "real" and "string" are indeed special cases of the Identifier because Identifier is defined as a letter followed by zero or more letters or digits and clearly "real" and "string" are 4 and 5 letters. However, the grammar EBNF of ML-subset also uses the terminals "<" and "<=" which the scanner would not see as a special case of Identifier. If we were to consider

the production rule from the point of view of the scanner, we would have RelationalOperator ::= "<" | "<" "=". Clearly that grammar would not be LL(1).

The Scanner that David A Watts proposes uses private methods such as "scanSeperator" and "scanToken" in a loop until the end of the stream is reached. As it reads in characters, it determines based on the first character after a separator what kind of token it should return. However, as we discussed above, since the lexicon grammar is not LL(1), his methods are not programmable in a systematic way.

Instead of using a method similar to EBNF for the lexicon, I decided to use regular expressions. Regexes were created to do pattern matching so this worked well. Each possible token, whether it be a terminal defined in the grammar's EBNF, or a nonterminal defined in the lexicon, is associated to a regular expression and an order. Creating a regex for each terminal defined in the EBNF is easy because those are simply strings whose regular expression is that same string. For example, the terminal in ML-subset "fun" matches the regular expression "fun". Another less trivial token such as Identifier is associated with the more complicated regex "[a-zA-Z]([a-zA-Z]|\\d)*" (this regex matches a character followed by zero or more characters or digits). Each token to regex association is listed out, one per line, in such a way that the bottom rules have priority over the top rules. For example, the Identifier rule is listed first and the rule for the "fun" token is listed below it. Hence, although the string "fun" matches both the regex "fun" and the one for the Identifier, the "fun" regex is given a higher priority so the token will be created correctly.

The scanner that I generate has a concept of "ignored" tokens vs "regular" tokens. Ignored tokens are things like separators and comments that the parser

will not actually use because it does not matter to the parser how much white space is in between tokens. As far as the parser is concerned, the source code "call fact(4);" is the same as "call fact (4) ;". However, in order to make things as generic as possible, I chose to have the user define explicitly what constitutes a separator and the scanner returns a token for each of these. The only inherent separator in the scanner is a new line character which must terminate a token. This is due to the limitation that each line of the source file is read one line at a time. Most languages, including ML, dictate that separators are white space such as spaces, tabs and new line characters but the only assumption my scanner makes is that new lines are separators, all others must be defined. Each "ignored" token is given a token kind of a negative value so it is easy for the parser to call the "scan" method until it retrieves a non-negative token.

The "TokenGenerator" class is responsible for generating both the Token and Scanner code for the Parser Generator. The input to the TokenGenerator is a lexicon InputStream (when run from the command line it would be a file but it could also be a text area on a web page) that is new line delimited. Lines that start with a "#" and blank lines are ignored. In addition to the blank lines and comments, the lexicon data has 3 sections which must be in the proper order. The first section is the ignored tokens optional section that must start with the line "#IGNORED TOKENS". Following that line is 0 or more ignored token rules that make up the separators. The token rules are the token name in all caps followed by a colon followed by a token definition. The token definition is either a regular expression enclosed in quotes or the name of a rule that will be defined in the third section. The second section contains the regular tokens and must start with the line "#TOKENS". Following that line are 0 or more token rules exactly like those in the first section. The last section must start with the line "#VARIABLES". This contains 0 or more lines of variable

rules that start with the variable name in all caps followed by a colon followed by a regular expression. Here is the token input for the tokens of the ML-subset language. You will recall that there are 34 tokens, 31 of which are exact string matches and 3 that are defined by regular expressions. You will also note that a token named "Seperator" is defined which tells the scanner that spaces and tabs separate tokens. Here is the lexicon for the ML-subset in this proprietary format:

```
#ML-subset Lexicon
#IGNORED TOKENS
SEPERATOR:Seperator

#TOKENS
IDENTIFIER:Identifier
INTEGERLITERAL:IntegerLiteral
STRINGLITERAL:StringLiteral
FUN:"fun"
INT:"int"
REAL:"real"
STRING:"string"
LIST:"list"
ELSE:"else"
IF:"if"
THEN:"then"
COLON:":"
SEMICOLON:";"
COMMA:","
NEG:"~"
NOT:"not"
LPAREN:"("
RPAREN:")"
LBRACKET:"["
RBRACKET:"]"
AND:"and"
ASTERISK:"*"
DIV:"/"
MOD:"mod"
INTDIV:"div"
EQUAL:"="
GREATER:">"
GEQ:">="
NEQ:"\="
LESS:"<"
LEQ:"<="
MINUS:"-"
OR:"or"
PLUS:"+"

#VARIABLES
```

```
IntegerLiteral:\d+
StringLiteral:'\w'
Identifier:[a-zA-Z]([a-zA-Z]|\d)*
Seperator:\x20|\t
```

Figure 3: ML-subset Token Rules

The user enters these rules in a text area in the Parser Generator GUI. The Parser Generator somehow needs to read each rule and load it in memory. It would be possible to create a scanner and a parser using the techniques discussed in this paper to scan each character and create an AST representing the lexicon. The scanner would read the string "FUN:fun" and create some sort of variable name token for the string "FUN" followed by a colon token corresponding to ":" and finally a third token for the regex "fun". The parser would then make sure that each line contains a variable name token followed by a colon token and finishing with a regex token. This would work fine but it is overkill for the task at hand. The lexicon format I developed was done in such a way that it is easy to parse using much simpler techniques. Each line above is either a comment line starting with "#" in which case it is ignored or it consist of a name/value pair (variable name / regular expression) separated by a colon. Hence the Parser Generator reads in these token rules using the following pseudo code:

```
List tokenRuleList = new ArrayList();
String line = null;
while ( (line = read next line) != null){
      if (line is new line or line starts with ("#"))
            ignore;
      else{
            int indexOfColon = line.indexOf(":");
            if (indexOfColon==-1)
                  throw error since colon not found;
            else{
                  String variableName = line.substring(0,indexOfColon);
                  String regex = line.substring(indexOfColon+1);
                  tokenRuleList.add( new TokenRule(variableName, regex);
            }
      }
}
```

Note the pseudo code does not address that the parser needs to know which section "#IGNORED TOKENS", "#TOKENS" or "#VARIABLES" it is in but that is a trivial programming task and does not help in the understanding of the basic parsing algorithm above.

Once the Parser Generator reads in the token rules for the scanner, it plugs them into a token template and a scanner template I created to generate the token and scanner classes. Hence the token and scanner that is generated for any language the user enters will be essentially the same as any other language except that the "scan()" method will use the token rules for the particular language in question. As we will see shortly, this differs from the parser for each language in that it is much more tailored for one particular language as the parser returns language-specific AST object trees and not more generic Token objects.

A simple, logical approach for the "scan()" method would be to read each character one at a time until a separator is reached and then see if the character sequence matches one of the token rule regular expressions. This would work if there were always a separator in between each type of token but that is not always the case. For example in the source code "num<=2" there are 3 tokens but no separators. In order to solve this problem, one might be tempted to change the "scan()" method so that it reads in each character at a time and checks each sequence as a new character is added if it follows one of the rules. If the sequence with the new character satisfies one of the rules, it continues until either a separator is met or a character that no longer makes the sequence satisfy the rule is met. It then takes the current sequence (minus the new character) and creates a token for the rule it matches. In our example above, it would read the first 3 characters 'n', 'u', 'm' and each time it added

the last character to the sequence, the Identifier rule would be satisfied. However, after reading the '<' character, the Identifier rule would no longer be satisfied so it would return the Identifier token "num" and start the next token with the character '<'. This new rule would work well for our sample source code but it is not generic enough. It is possible to have a token rule that uses a regular expression that will not be satisfied until the last character is read. For example, suppose the token rule for *Identifier* was changed to [a-zA-Z]([a-zA-Z]|\d)*[a-zA-Z] which is a letter followed by 0 or more letters or numbers and finally terminated by a letter. The string "a123456b" would be a valid *Identifier*. However, using the rule above we would return three tokens "a", "123456" and "b" because the character sequence "a1" would not satisfy any rules so the token "a" would first be returned (assuming "a" did match another token rule).

The implementation "scan()" method in the scanner that the Parser Generator generates is not completely intuitive at first but it solves the problems discussed above. Instead of reading character by character and seeing if the sequence matches each time, I read the entire string and see if that matches. If not, I look at the substring starting at the second character to find a match and so forth until I am only looking at the last character if not rules have been matched. This algorithm in pseudo Java code is as follows (see Appendix J for the exact code):

```
List tokenList = new ArrayList();
String line = readEntireLine();
String linecopy = line;
while (linecopy is not empty){
      if (linecopy matches one of the token rules){
            tokenList.add(token corresponding to linecopy);
            linecopy = line minus the linecopy that matched at the end;
      }
}
tokenList.reverseOrder();
```

This algorithm finds the last token first and makes it way back to the first token. Hence the token list must be reversed once the list of tokens is calculated. The advantage of this approach is that the scanner can be very generic and easily used by different languages by just replacing the method that checks if the linecopy matches one of the token rules (since each language has its own set of token rules). Our sample string "num<=2" is handled by this algorithm by checking the strings "num<=2" against each production rule and seeing there is no match so then checking "um<=2" and so forth until it finally finds a match with "2". It then looks for matches with "num<=", "um<=", "m<=" until it finds a match at "<=". Then the remaining string "num" matches a token rule. Token rules that match more complicated regular expression such as the changed Identifier we used above would also work. One of the main disadvantages to this approach is potentially poor performance because of all the match checks that are made for each substring. If you have a string s containing l characters, you could have worst case (l*(l+1))/2 match checks if each character in the string was its own token. For example, the string "1+2+3+4+5" which contains 9 characters (and 9 tokens) would make the scanner do 9+8+7+6+5+4+3+2+1 = (9*(9+1))/2 = 45 match checks.

Parser Implementation

Generating a parser is more complicated than creating a scanner. The first issue that makes this so is that when generating the scanner we could use regular expressions because we were dealing with characters, but the parser no longer deals with characters but with tokens. Secondly, we can not directly use the EBNF grammar the user entered to create the parser because the production rule definition in an EBNF grammar can be too complicated. To simplify if, the grammar needs to be transformed into an AST grammar first. Lastly, the parser returns an AST tree which is a data structure that is much more complicated than the Token which is returned by the scanner.

Reading the user-entered EBNF grammar in memory

The user enters the grammar for which they want a parser auto-generated. The Parser Generator needs to load this user-entered grammar in memory. What better way to do this than to create a scanner and parser for an EBNF grammar? This is precisely the approach that I took. Of course, if I already had the Parser Generator, I would be able to automatically generate the scanner and parser that I needed. I used the techniques described later in this section to manually create a scanner and a parser for the input EBNF grammar. In order to write a parser for an EBNF grammar, I had to express the EBNF syntax in EBNF notation just like we earlier expressed the ML-subset in EBNF notation:

```
Grammar           ::= ProductionRuleList .
ProductionRuleList::= ProductionRule {ProductionRuleList} .
ProductionRule    ::= NonTerminal "::=" Definition ".".
Definition        ::= DefinitionSequence { "|" DefinitionSequence } .
DefinitionSequence::= DefinitionBody DefinitionSequence .
DefinitionBody    ::= Symbol |
```

```
                          "(" Definition ")" |
                          "[" Definition "]" |
                          "{" Definition "}".
Symbol            ::= Terminal | NonTerminal .
NonTerminal       ::= Identifier .
Terminal          ::= """ String """.
```

Figure 4: EBNF of EBNF notation.

As you can see, this grammar defines a language that can have 1 or more *ProductionRules* that are in the form of "NonTerminal ::= Definition ." where the definition can contain alternative symbols as well as grouping and empty grouping symbols.

From the EBNF grammar of EBNF notation, I derived the AST grammar of the EBNF notation using the techniques that I will describe below:

```
Grammar                 ::= ProductionRuleList .
ProductionRuleList      ::= ProductionRule ProductionRuleListTail .
ProductionRuleListTail  ::= ProductionRuleList ProductionRuleListTail .
ProductionRule          ::= NonTerminal Definition .
Definition              ::= DefinitionSequence DefinitionTail .
DefinitionTail          ::= DefinitionSequence DefinitionTail .
DefinitionSequence      ::= DefinitionBody DefinitionSequence .
DefinitionBody          ::= SymbolDefinition |
                            GroupingDefinition |
                            ZeroOrOneDefinition |
                            ZeroOrMoreDefinition .
SymbolDefinition        ::= Symbol.
GroupingDefinition      ::= Definition .
ZeroOrOneDefinition     ::= Definition .
ZeroOrMoreDefinition    ::= Definition .
Symbol                  ::= TerminalSymbol |
                            NonTerminalSymbol .
TerminalSymbol          ::= Terminal .
NonTerminalSymbol       ::= NonTerminal.
NonTerminal             ::= Identifier .
Terminal                ::= String .
```

Figure 5: AST of EBNF notation.

As a result, the AST tree's top node will be a Grammar object that contains a ProductionRuleList. The ProductionRuleList contains a ProductionRule and a ProductionRuleListTail, and so forth. Given the EBNF and AST grammars of EBNF

notation, I then manually wrote a scanner and parser capable of reading in any grammar expressed in EBNF notation. When the ML-subset grammar is read in by EBNF notation parser, it is loaded in memory in the following tree:

```
ebnfast.Grammar
 ebnfast.ProductionRuleList
  ebnfast.ProductionRule
   ebnfast.NonTerminal
   ***Program***
   ebnfast.Definition
    ebnfast.DefinitionSequence
     ebnfast.ZeroOrMoreDefinition
      ebnfast.Definition
       ebnfast.DefinitionSequence
        ebnfast.SymbolDefinition
         ebnfast.NonTerminalSymbol
          ebnfast.NonTerminal
          ***Functions***
        ebnfast.DefinitionSequence
         ebnfast.SymbolDefinition
          ebnfast.TerminalSymbol
           ebnfast.Terminal
           ***;***
     ebnfast.DefinitionSequence
      ebnfast.SymbolDefinition
       ebnfast.NonTerminalSymbol
        ebnfast.NonTerminal
        ***Call***
```

Figure 6: Partial listing of ML-subset grammar in AST tree.

The entire listing can be found in Appendix E. The Parser Generator then manipulates this in-memory representation of the ML-subset grammar.

Converting EBNF grammars to AST grammars

EBNF production rules can be very complicated because they may contain many combinations of groupings and alternatives. The AST grammar is much simpler because it imposes the following restrictions:

1. The grouping tags "(" and ")" are not allowed.

2. Alternatives are allowed, however an alternative must only contain one nonterminal or terminal. Alternatives cannot be sequences.
3. The zero or one tags "[" and "]" are not allowed.
4. The zero or more tags "{" and "}" are not allowed.

To transform an EBNF grammar to an AST grammar, we take the following steps:

1. Sequences:

Sequences are allowed in the AST grammar. If an EBNF rule consists solely of a sequence, then the AST rule is the same as the EBNF rule.

Suppose your EBNF was:

```
A::= B C D.
```

This is valid AST so your new AST consists of:

```
A::= B C D.
```

2. Groupings:

Groupings are not allowed in AST however it is trivial to replace a grouping with a new production rule.

Suppose your EBNF was:

```
A::= (B C).
```

To create the AST create a new production rule for (B C):

```
A           ::= ABGroup.
ABGroup     ::= B C.
```

3. Alternatives:

Alternatives are allowed in the AST grammar as long as each alternative consists of a single nonterminal or terminal. To remove any possible grouping, we treat each alternative as a group and replace them by a new production rule.

Suppose your EBNF was:

```
A::= B | C D.
```

To create the AST pretend the EBNF rule was A::= (B) | (C D) and create a new rule for (B) and for (C D):

```
A           ::= AB | AC.
AB          ::= B.
AC          ::= C D.
```

4. Zero or One:

A zero or one in an EBNF is treated the same way as a group. A new production rule is created for everything that was in the brackets.

Suppose your EBNF was:

```
A::= [B C].
```

To create the AST create a new production rule for (B C):

```
A           ::= ABZeroOrOne.
ABZeroOrOne ::= B C.
```

5. Zero or More:

A zero or more is treated almost the same way as a grouping, but the new production rule refers to itself to create a structure that can have many elements.

Suppose your EBNF was:

```
A::= { B }.
```

To create the AST create a new production rule for (B C):

```
A           ::= ABZeroOrMore.
ABZeroOrMore::= B ABZeroOrMore.
```

As you can see, converting an EBNF grammar to an AST grammar generally adds rules. Understanding the rules used to make the transformation will allow you to modify the initial EBNF grammar so that the resulting AST grammar is not too foreign. For example, avoiding the use of groupings or replacing groupings with a new production rule in the EBNF grammar will make the change unnecessary when converting it to AST. Here is the AST for the ML-subset:

```
Program ::= ProgramFunctionsZeroOrMore Call ";" .
ProgramFunctionsZeroOrMore ::= Functions ";" ProgramFunctionsZeroOrMore .
Functions ::= "fun" Identifier "(" FunctionsParListZeroOrOne ")" ":" Type
"=" Expression .
FunctionsParListZeroOrOne ::= ParList .
ParList ::= Par ParList_commaZeroOrMore .
ParList_commaZeroOrMore ::= "," Par ParList_commaZeroOrMore .
Par ::= Identifier ":" Type .
Type ::= SimpleType Type_listZeroOrOne .
Type_listZeroOrOne ::= "list" .
SimpleType ::= "int" | "real" | "string" .
Expression ::= ExpressionConditionalExpression | ExpressionBasicExpression
.
ExpressionConditionalExpression ::= ConditionalExpression .
ExpressionBasicExpression ::= BasicExpression .
ConditionalExpression ::= "if" BasicExpression "then" Expression "else"
Expression .
BasicExpression ::= PrimaryExpression
BasicExpressionPrimaryOperatorZeroOrMore .
BasicExpressionPrimaryOperatorZeroOrMore ::= PrimaryOperator
PrimaryExpression BasicExpressionPrimaryOperatorZeroOrMore .
PrimaryOperator ::= "or" | "and" .
PrimaryExpression ::= SimpleExpression
PrimaryExpressionRelationalOperatorZeroOrOne .
PrimaryExpressionRelationalOperatorZeroOrOne ::= RelationalOperator
SimpleExpression .
RelationalOperator ::= "<" | "<=" | ">" | ">=" | "=" | "\=" .
SimpleExpression ::= SimpleExpression_negZeroOrOne Term
SimpleExpressionAddingOperatorZeroOrMore .
SimpleExpression_negZeroOrOne ::= "~" .
SimpleExpressionAddingOperatorZeroOrMore ::= AddingOperator Term
SimpleExpressionAddingOperatorZeroOrMore .
AddingOperator ::= "+" | "-" .
Term ::= Factor TermMultiplyingOperatorZeroOrMore .
TermMultiplyingOperatorZeroOrMore ::= MultiplyingOperator Factor
TermMultiplyingOperatorZeroOrMore .
MultiplyingOperator ::= "*" | "/" | "div" | "mod" .
Factor ::= Factor_integerliteral | Factor_stringliteral |
Factor_identifier | FactorListLiteral | Factor_lparen | Factor_not .
Factor_integerliteral ::= IntegerLiteral .
Factor_stringliteral ::= StringLiteral .
```

```
Factor_identifier ::= Identifier Factor_identifier_lparenZeroOrOne .
Factor_identifier_lparenZeroOrOne ::= "("
Factor_identifier_lparenZeroOrOneArgListZeroOrOne ")" .
Factor_identifier_lparenZeroOrOneArgListZeroOrOne ::= ArgList .
FactorListLiteral ::= ListLiteral .
Factor_lparen ::= "(" Expression ")" .
Factor_not ::= "not" Factor .
ListLiteral ::= "[" ListLiteralExpressionListZeroOrOne "]" .
ListLiteralExpressionListZeroOrOne ::= ExpressionList .
ExpressionList ::= Expression ExpressionList_commaZeroOrMore .
ExpressionList_commaZeroOrMore ::= "," Expression
ExpressionList_commaZeroOrMore .
Call ::= Identifier "(" CallArgListZeroOrOne ")" .
CallArgListZeroOrOne ::= ArgList .
ArgList ::= Expression ArgList_commaZeroOrMore .
ArgList_commaZeroOrMore ::= "," Expression ArgList_commaZeroOrMore .
```

Figure 7: AST Listing for ML-subset grammar.

Creating AST Java classes from an AST grammar

An AST grammar only contains production rules that are defined as sequences (e.g. A::= A1 A2 … AN) or as alternatives of single nonterminals (e.g. B::= B1 | B2 | … | BN). We can apply a standard set of rules for each kind of production rule to generate the AST Java objects. Here are the steps:

1. Create an abstract class Ast.java that all other generated AST objects will extend. The only abstract method defined is used as a hook to the visitor pattern (see "Visitor" section).

```
public abstract class Ast{
      public abstract Object visit(Visitor v, Object arg);
}
```

2. For production rules that are defined as alternatives B::= B1 | B2 | … | BN, create a single abstract class for B that each alternative will extend when they are generated later. This superclass B will allow us to have B

variables that point to instances of B1, …, BN and loop through them using polymorphism.

```
public abstract class B extends Ast{}
```

3. For production rules that are defined as a sequence A::= A1 A2 … AN, there can be two cases: either A was an alternative in another production rule or it was not. These classes contain the data representing the source code so they are the most complicated.

```
public class A{
      public A1 a1;
      …
      public AN an;
      public int line;

      public A(A1 a1, …, AN an, int line){
            this.a1 = a1;
            …
            this.an = an;
            this.line = line;
      }

      public Object visit(Visitor v, Object arg){
            return v.visitA(this, arg, line);
      }
}
```

If A had been an alternative of some other production rule, then it would extend that class. For example, if before the A production rule we had had B::= A | X. Then A.java would look like:

```
public class A extends B{
      //everything else is the same
}
```

Java does not support multiple inheritance so there would be a problem if you ever had a scenario where a nonterminal was an alternative in 2 production rules as follows:

```
N ::= A | B
M ::= A | C
```

This could be the situation in the original user-entered EBNF, however these 2 rules would have been converted when the EBNF converted to an AST. See rule #3 Alternatives in the "Convert EBNF grammars to AST grammars" section to see that the AST production rules equivalent to the EBNF rules above would be:

```
N ::= NA | NB.
NA::= A.
NB::= B.
M ::= MA | MC.
MA::= A.
MC::= C.
```

Therefore we never have a situation where the same nonterminal is in more than one production rule alternative so an AST class never has to extend more than one class.

4. Create the Visitor.java that will have one visitA method for each object A that was created as a result of being defined by a sequence.

```
public interface Visitor{
      public Object visitA(A a, Object arg, int line);
      public Object visitA1(A1 a1, Object arg, int line);
      …
      public Object visitAN(AN an, Object arg, int line);
}
```

Generating a parser from an AST grammar

Just as there is a systematic way to create the AST objects given the AST grammar, there is a systematic way to create the parser:

1. Create a public parse() method that will call the private parseProgram() method and will return the top node Program of the AST tree:

```
public Program parse(){
            SourceFile sourceFile = new SourceFile();
```

```
                  scanner = new Scanner(sourceFile.openFile());
                  currentToken = (Token) scanner.scan();
                  Program program = parseProgram();
                  if(currentToken.getKind() != Token.EOT)
                        new Error("Syntax error: Redundant characters at
      the end of program.",currentToken.line);
                  return program;
                  }
```

2. Create 2 methods acceptIt() and accept() that will get the next token from the scanner. The acceptIt() method is called when you already tested the current token and you know it is valid. The accept() method takes a token type as a parameter and checks the current token against that type before getting the next token from the scanner:

```
      private void accept(byte expectedKind){
            if(currentToken.getKind() == expectedKind)
                  currentToken = (Token) scanner.scan();
            else
            new Error("Syntax error: " + currentToken.spelling + " is not
expected.  Expected "+expectedKind+" but got
"+currentToken.getKind()+".",currentToken.line);
      }

      private void acceptIt(){
            currentToken = (Token) scanner.scan();
      }
```

3. Loop through all the production rules. You will recall that each production rule definition is either a sequence of terminals or nonterminals or a series of alternatives each containing a single terminal or nonterminal.

4. For each production rule A, create a method that will return an AST object of type A whose name is parseA as follows:

```
      private A parseA(){
            //refine
      }
```

5. If the definition of A is an alternative, that is A::= A1 | A2 | … | An, refine the body as follows:

```
switch (currentToken.kind){
      case starters(A1):
            A1 a1 = null;
            a1 = parseA1();
            return a1;
      …
      case starters(An):
            An an = null;
            an = parseAn();
            return an;
      default:
            new Error("Syntax error.  Expected <list expected
elements>.  Got "+currentToken.getKind()+" spelled
"+currentToken.getSpelling()+".", currentToken.line);
      return null;
}
```

Looking at this rule, you can now understand why the grammar needs to be LL(1). According to rule 1 of LL(1) grammars, starters(Ai) and starters(Aj) must be different for each i,j where i≠j. If this were not the case, then there would be multiple "case" statements with the same value. This would be a problem because the program would not know which case to execute and, in fact, in Java it would produce a compilation error.

6. If the definition of A is a sequence, that is A::= A1 A2 … An, refine the body as follows depending on the type of Ai for each i:

   a. If Ai is a terminal, refine the body as:

```
String _aiSpelling = null;
_aiSpelling = currentToken.spelling;
accept(Token.AI);
```

   b. If Ai is a nonterminal that must occur once (that is, it is not a zero or one or a zero or more), refine the body as:

```
Ai ai = null;
ai = parseAi();
```

c. If Ai is a nonterminal that can occur zero times, refine the body as:

```
Ai ai = null;
if (currentToken.kind in starters(Ai)){
      ai = parseAi();
}
```

Looking at this rule, you can now understand a second reason the grammar needs to be LL(1). If the starter set of the zero or more or zero or one grouping is not disjoint from the starter set of the following element, then the "if" condition will be true in some cases when the zero grouping is not empty.

At the end of the method body, you must return a new object of type A that you create, passing all the instance variables that you instantiated above:

```
return new A(instance variables such as _aiSpelling and
ai,currentToken.line);
```

We will now look at 2 examples encompassing each of these rules:

Example: In ML-subset, the Expression nonterminal is defined as Expression ::= ExpressionConditionalExpression | ExpressionBasicExpression. From the starter set table 2, we know that starters(ExpressionConditionalExpression) = starters(ConditionalExpression) = { "if" } and starters(ExpressionBasicExpression) = starters(BasicExpression) = { IntegerLiteral, "(", "not", StringLiteral, "[", Identifier, "~" }.

Following rules 1 and 2 above, the parse method for Expression is:

```
private Expression parseExpression(){

      switch (currentToken.kind){
            case Token.IF:
ExpressionConditionalExpression ExpressionConditionalExpression = null;
ExpressionConditionalExpression = parseExpressionConditionalExpression();
            return ExpressionConditionalExpression;
            case Token.INTEGERLITERAL:
            case Token.NOT:
            case Token.LBRACKET:
            case Token.NEG:
            case Token.STRINGLITERAL:
            case Token.LPAREN:
            case Token.IDENTIFIER:
ExpressionBasicExpression ExpressionBasicExpression = null;
ExpressionBasicExpression = parseExpressionBasicExpression();
            return ExpressionBasicExpression;
            default:
            new Error("Syntax error.  Expected
ExpressionConditionalExpression or ExpressionBasicExpression.  Got
"+currentToken.getKind()+" spelled "+currentToken.getSpelling()+".",
currentToken.line);
      return null;
      }
}
```

Example: In ML-subset, the ExpressionList_commaZeroOrMore ::= "," Expression

ExpressionList_commaZeroOrMore .

```
private ExpressionList_commaZeroOrMore
parseExpressionList_commaZeroOrMore(){

      String _commaSpelling = null;
      _commaSpelling = currentToken.spelling;
      accept(Token.COMMA);

      Expression expression = null;
      expression = parseExpression();

      ExpressionList_commaZeroOrMore expressionList_commaZeroOrMore =
null;
      if (currentToken.kind==Token.COMMA){
            expressionList_commaZeroOrMore =
      parseExpressionList_commaZeroOrMore();
      }

return new
ExpressionList_commaZeroOrMore(_commaSpelling,expression,expressionList_co
mmaZeroOrMore,currentToken.line);
}
```

The parser that is generated using this technique is a recursive-descent parser which is a top-down parsing algorithm. "A recursive-descent parser for a grammar G consists of a group of methods parseN, one for each nonterminal symbol N of G. The task of each method parseN is to parse a single N-phrase. There parsing methods cooperate to parse complete sentences."[v] Since *Program* is the first nonterminal symbol in the AST grammar, the parser is started by calling the parseProgram() method. That method in turn delegates to the parseProgramFunctionsZeroOrMore() method if it detects the Token.FUN and then to the parseCall() method. A method such as parseProgramFunctionsZeroOrMore() can call itself recursively to parse the next function declarations. As each parseN() method calls another parseM() method, the AST tree gets deeper and deeper.

The Visitor Pattern

The visitor pattern is the most advanced programming technique used in the parser code and therefore deserves a thorough explanation. A design pattern is a solution to a specific problem. Well known patterns include "singleton", "factory", "façade" and "strategy". "In general, you need a Visitor when you have numerous algorithms to run on the same heterogeneous object structure." The AST objects in my project meet both of the main criteria: the objects are heterogeneous and you may need to traverse the AST tree several times.

Although some AST objects may be homogeneous by extending the same subclass or, theoretically, by implementing the same interface (in practice this is not the case because no interfaces are used when defining the AST objects), as a collection not all of the objects will be uniform. For example, if you look at the EBNF AST objects, you will see that 4 of the AST objects, namely the "GroupingDefinition", "SymbolDefinition", "ZeroOrMoreDefinition" and "ZeroOrOneDefinition" all extend the same "DefinitionBody". So these 4 objects are homogeneous. However, other objects in the EBNF AST share no such parent. For example, although "TerminalSymbol" and "NonTerminalSymbol" extend the "Symbol" object, there is no link to the "DefinitionBody". Hence more straightforward methods of traversing the AST tree such as polymorphism will not work.

Being able to traverse an AST tree for different algorithms is extremely useful. In fact, the program I wrote that transforms an EBNF grammar into an AST grammar traverses the EBNF AST tree several times. I have a class named "StarterSetCheck" that visits each EBNF AST to create a table containing the starter set for each production rule. There is another class

"EBNF2ASTConverter" that traverses the same EBNF AST tree and transforms the EBNF AST objects into AST AST objects. Both implement the same Visitor interface but do something different.

A Visitor implementation is somewhat complicated but understanding it is well worth the effort since it will make much of the code significantly clearer. Here are the steps:

- Create a "Visitor" interface that has a method for each object you will want to visit. The method will take as argument an instance of the object you are visiting.

```
public interface Visitor{
      public visitB(B b);
      public visitC(C c);
}
```

- Optionally, create an abstract class that contains an abstract method named "visit" that takes an input a "Visitor" instance and returns an object. The "Ast.java" class plays this role.

```
public abstract class A{
      public abstract Object visit(Visitor visitor);
}
```

- Create your classes that extend the abstract class and overwrite the "visit" method by calling the appropriate method in the "Visitor" class and passing itself as input. This double-dispatch feature is the key to the pattern. It gives the Visitor access to the object without needing to know how to get that object.

```
public class B extends A{
      public C c = new C();

      public Object visit(Visitor visitor){
```

```
            return visitor.visitB(this);
      }
}

public class C extends A{
      public Object visit(Visitor visitor){
            return visitor.visitC(this);
      }
}
```

- Implement the Visitor by creating a class that implements the "Visitor" interface and does something.

```
public class Algorithm implements Visitor{
            public visitB(B b){
                  //do something with b
                  b.c.visit(this);
            }
            public visitC(C c){
                  //do something with c
            }
}
```

That is the basic Visitor implementation. You will see that as long as the implementation of each visitX method calls the visitY method for each object Y contained in X, then a visit method will have been called for object in your object tree. Note that in the code output by the Parser Generator, the "visit" methods actually take 2 parameters: the Visitor as described above and an arg Object. The callback methods also pass additional data: they pass themselves, the arg object that had been an input and a line number.

Performance

When writing the Parser Generator, performance was not a concern. My goal was to write the code regardless of the efficiency. I knew I could always change the implementation of certain methods or classes to optimize them later. However, after the Parser Generator was done, I was curious as to how efficient it was. Instead of looking at the performance of the Parser Generator itself, I was more interested in the performance of the scanner and parser that it generated. The logic behind this is that the Parser Generator would only be run once per grammar but the generated scanner and parser could be run many times with different source programs and could be the starting point for compilers or other extensions to the parser that takes an AST tree as input.

Unfortunately, I do not have access to any code that measures the parsing speed of ML-subset against which to compare my code. Luckily, I do know one other grammar for which there are many accessible parsers: XML. XML, or Extensible Markup Language, is a markup language that looks very similar to HTML. However, instead of conveying display logic like HTML, XML is used to represent data and metadata. An XML file is a text file that contains element tags and attributes around data. Here is a sample XML:

```
<person>
      <firstname>Josh</firstname>
      <lastname>Foure</lastname>
</person>
```

There are many tutorials online that go in much greater depth[10]. Here is the XML EBNF Grammar we will be using:

```
Program ::= Element.
Element ::= "<" Word ">" [Data] {Element} [Data] "</" Word ">".
Data::= Word {" " Word}.
```

It is easy to see that this grammar is LL(1) and leave that as an exercise to the reader. If you are familiar with XML, you will notice that the XML EBNF I propose does not cover all possible XML structures. For example, the XML declaration that contains the version and the character encoding is omitted altogether as are attributes. Another interesting point is that in XML the begin tag name must match the end tag name. In EBNF terms, this means that the first Word after the "<" must match the last Word after the "</". However, this is a contextual rule, not a syntactical rule. Since all third party XML parsers do this contextual analysis and mine does not, it is not an exact test. However, I am interested mainly in orders of magnitude.

There are 2 well known methods of parsing XML in Java: SAX and DOM. A SAX, or Simple API for XML, parser goes through an XML file and calls callback methods when events such as encountering a beginning or end of an element are reached. It does not create any sort of in-memory structure so it is not a good candidate to compare with my parser that creates an AST. DOM, or Document Object Model, reads an entire XML file into memory and returns a Document object which contains a tree of Node objects. This representation is roughly equivalent to Program and Element AST objects that the Parser Generator will generate given the EBNF above. A key difference between our AST tree and a DOM Node structure is how we handle sequences of elements. In AST, because of how we defined zero or more groupings, a list of 5 elements would look like an unbalanced tree that is five levels deep. In DOM, a list object is used to all 5 elements are at the same level which gives you a tree with less depth and more breadth.

Because our parser generates an AST tree and the DOM parser generates a tree containing lists, we will look at 2 types of XML structures that mimic how

each parser works internally. We will first look at "sibling" entries which are lists of elements and look something like this:

```
<personlist>
<person>
 <firstname>Josh0</firstname>
 <lastname>Foure0</lastname>
</person>
<person>
 <firstname>Josh1</firstname>
 <lastname>Foure1</lastname>
</person>
…
<person>
 <firstname>JoshN</firstname>
 <lastname>FoureN</lastname>
</person>
</personlist>
```

We will also look at a "tree" structure where each person element contains a child person element as can be seen:

```
<personlist>
<person>
 <firstname>Josh0</firstname>
 <lastname>Foure0</lastname>
 <person>
  <firstname>Josh1</firstname>
  <lastname>Foure1</lastname>
  …
  <person>
   <firstname>JoshN</firstname>
   <lastname>FoureN</lastname>
  </person>
 </person>
</person>
</personlist>
```

We will look at files that contain 1, 10, 100, 1000 and 10000 entries.

	Parser Generator	**DOM Parser**
1 Sibling Entry	50	10
10 Sibling Entries	110	10
100 Sibling Entries	441	50
1000 Sibling Entries	4757	240
10000 Sibling Entries	java.lang.StackOverflowError	391
1 Tree Entry	10	10
10 Tree Entries	50	10

100 Tree Entries	430	10
1000 Tree Entries	java.lang.StackOverflowError	30
10000 Tree Entries	java.lang.StackOverflowError	280

Table 6: Response time in ms for 2 types of XML files.

As the table indicates, the scanner and parser that the Parser Generator generates performs much more poorly that the Sun's DOM Parser. This is not completely unexpected as the Parser Generator creates a very generic scanner and parser whereas the Sun Parser is very specific to XML and has been optimized for mission critical real time applications. I also strongly suspect that the scanner I generate, which parses the whole line from end to start looking for the biggest token matches, is not as efficient as it could be. An interesting next step would be to optimize the scanner to see how much of the performance problems were due to that and how much was due to the parsing logic.

Thesis Run Guide

The Parser Generator program consists of 111 Java files in 16 packages. You can see a brief description of the packages and the entire code in the appendixes. The code is packaged in a jar file called JoshFoureThesis2004.jar. In addition to this jar, a 3rd party jar that I did not write called "log4j-1.2rc1.jar" is included. This jar is part of the Apache Jakarta Project[9] which let me debug my program much faster than I would have using a debugger or sprinkling System.out.println calls in my files. "With log4j it is possible to enable logging at runtime without modifying the application binary. The log4j package is designed so that these statements can remain in shipped code without incurring a heavy performance cost. Logging behavior can be controlled by editing a configuration file, without touching the application binary."[vi]

You can run the code by extracting the JoshFoureThesis2004.jar and placing the log4j-1.2rc1.jar in the root directory. Suppose you extract the JoshFoureThesis2004.jar in a location called %THESISPATH%. You then run the Parser Generator by going into %THESISPATH% (i.e. "cd %THESISPATH%") and running the following command:

```
java -cp .;log4j-1.2rc1.jar application.Main
```

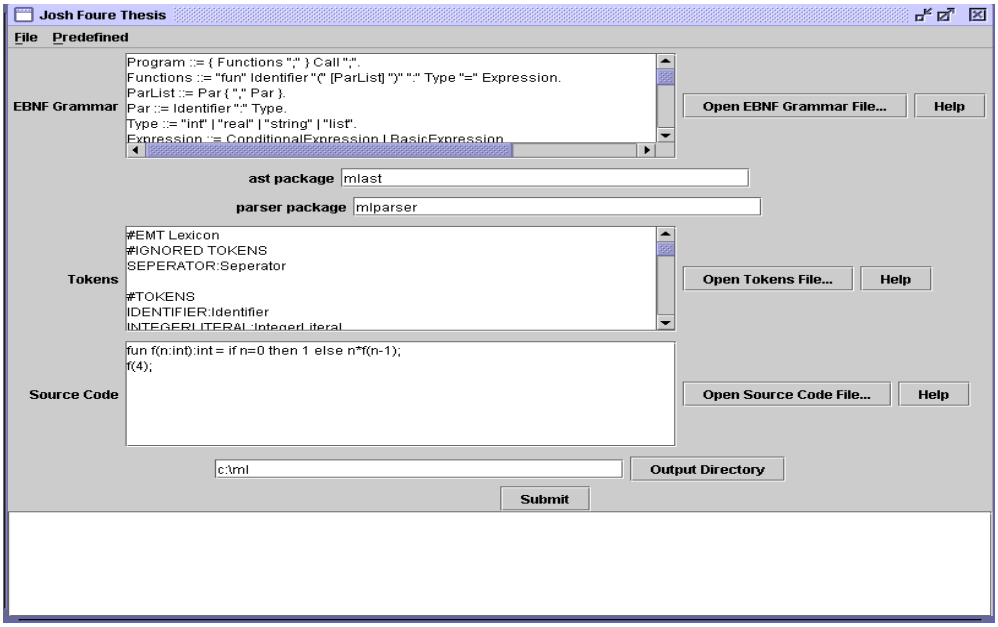


Figure 8: Screen shot of Parser Generator GUI.

The "File" menu lets you save your inputs in a file or load a file you have previously saved. The "Predefined" menu lets you pick one of 4 languages and fills out the text fields appropriately for that language. The 4 languages are: EMT, PL, Pascal and Badii (a language created by Professor Badii's son and used as a test). The other languages discussed in this paper such as ML-subset and XML are not programmed into the GUI but they can be copied and pasted easily from the appendix.

The Parser Generator takes 6 inputs. We have already discussed the EBNF format that the grammar is entered in and the proprietary format for the token rules. In addition to this information, the user must enter the package name

for the AST classes that will be generated. The user also enters the package name for the parser (and scanner) that will be generated. The fifth input the user enters is a sample source code. The source code is completely ignored by the Parser Generator when it creates the scanner and the parser. The only thing the Parser Generator does with the source code is output it in a file called "test.txt" that can be used by the user to test the generated scanner and parser. Naturally, the generated scanner and parser works for every single possible program that is valid for the given language, not just the one entered by the user. The last input the user enters is the directory that all of the generated code will be generated in.

Running the Generated Code

The Token.java, Scanner.java and Parser.java are generated in the parser's package. The Visitor.java and all the AST classes are generated in the ast's package. In addition to all the code generated, 2 batch files are created in the output directory: compile.bat and run.bat. The compile batch file runs the javac compiler to compile all the generated .java files. The run.bat runs the parser with the sample source code the user entered in the GUI. Here is the output of the generated ML-subset parser with the sample program that calculates the factorial:

```
mlast.Program
  mlast.ProgramFunctionsZeroOrMore
    mlast.Functions
      <fun>
      mlast.Identifier
      ***fact***
      <(>
      mlast.FunctionsParListZeroOrOne
        mlast.ParList
          mlast.Par
            mlast.Identifier
            ***n***
            <:>
```

```
            mlast.Type
              mlast.SimpleType
          <)>
          <:>
          mlast.Type
            mlast.SimpleType
          <=>
          mlast.ExpressionConditionalExpression
            mlast.ConditionalExpression
              <if>
              mlast.BasicExpression
                mlast.PrimaryExpression
                  mlast.SimpleExpression
                    mlast.Term
                      mlast.Factor_identifier
                        mlast.Identifier
                        ***n***
                  mlast.PrimaryExpressionRelationalOperatorZeroOrOne
                    mlast.RelationalOperator
                    mlast.SimpleExpression
                      mlast.Term
                        mlast.Factor_integerliteral
                          mlast.IntegerLiteral
                          ***0***
              <then>
              mlast.ExpressionBasicExpression
                mlast.BasicExpression
                  mlast.PrimaryExpression
                    mlast.SimpleExpression
                      mlast.Term
                        mlast.Factor_integerliteral
                          mlast.IntegerLiteral
                          ***1***
              <else>
              mlast.ExpressionBasicExpression
                mlast.BasicExpression
                  mlast.PrimaryExpression
                    mlast.SimpleExpression
                      mlast.Term
                        mlast.Factor_identifier
                          mlast.Identifier
                          ***n***
                        mlast.TermMultiplyingOperatorZeroOrMore
                          mlast.MultiplyingOperator
                          mlast.Factor_identifier
                            mlast.Identifier
                            ***fact***
                            mlast.Factor_identifier_lparenZeroOrOne
                              <(>

mlast.Factor_identifier_lparenZeroOrOneArgListZeroOrOn
      e
                                    mlast.ArgList
                                      mlast.ExpressionBasicExpression
                                        mlast.BasicExpression
                                          mlast.PrimaryExpression
                                            mlast.SimpleExpression
```

```
                                                mlast.Term
                                                  mlast.Factor_identifier
                                                    mlast.Identifier
                                                    ***n***

mlast.SimpleExpressionAddingOperatorZeroOr
      More
                                                  mlast.AddingOperator
                                                  mlast.Term
                                                    mlast.Factor_integerliteral
                                                      mlast.IntegerLiteral
                                                      ***1***
                                <)>
        <;>
        mlast.Call
          mlast.Identifier
          ***fact***
          <(>
          mlast.CallArgListZeroOrOne
            mlast.ArgList
              mlast.ExpressionBasicExpression
                mlast.BasicExpression
                  mlast.PrimaryExpression
                    mlast.SimpleExpression
                      mlast.Term
                        mlast.Factor_integerliteral
                          mlast.IntegerLiteral
                          ***4***
          <)>
        <;>
```

Figure 9: Output of ML-subset parser

Future enhancements

As with any project that lasts one year in duration and is over 200 pages of code, I found a few things that I would like to change or tweak if I had no time constraints. As you write more and more code, you start seeing patterns emerge which presents opportunities to refactor your code in a more efficient way. Below I discuss six possible future enhancements:

1. Optimize the scanner: The goal of this project was to automate parser generation. However, it is not possible to test a parser without a scanner so the generation of a scanner became a necessity, not a goal. Generating a scanner follows many of the same steps as generating a parser with one key difference: scanner grammars are not LL(1).

2. Simplify AST grammars: The ML language that is used as an example throughout this paper has 22 rules in its EBNF. However, the algorithm I used to create an AST grammar balloons the number of production rules to 44 rules. Many of the new rules come from the fact that there are 13 groupings in the original EBNF that can be empty (either zero or more or zero or one) and each of these gets a new production rule in the AST. Another source of additional production rules comes from the Factor nonterminal which is defined by 6 alternatives which adds 6 more AST production rules. I have already implemented code that does not add additional production rules if each alternative is a single terminal (e.g. AddingOperator and MultiplyingOperator) but additional optimization may be possible.

3. Performance analysis: As discussed in the "performance" section, the scanner and parser code that is generated is not necessarily efficient. It would be interesting to improve the scanner especially and compare the efficiency with parsers used for mission critical applications. Looking at how javac parses source code may be a starting point although the EBNF of Java is probably quite complicated.

4. Error handling: The scanner and parsers that I used to read in the user's EBNF and base the output of the Parser Generator on did not use the Java exception handling. When an error is reached an "Error" object is instantiated and printed out and the program is halted. This works sufficiently well when you run an application from a command line but not from a GUI where you need to report the error, or if you want to extend the scanner or parser. As a result, I started adding standard Java exception handling where Exceptions would be thrown and caught by the GUI program to display the error message. Unfortunately, there are still areas in the code that don't use this improved error handling logic.

5. Grammar manipulation: Although the user-entered grammar is validated to make sure that each nonterminal used is defined and starter sets are calculated, there is no LL(1) check (the user must enter an LL(1) grammar by contract). Writing an algorithm to perform the validation and/or doing some of the transformations discussed earlier to convert the user-entered language into LL(1) if it is not already would be an interesting enhancement.

6. New features: The Parser Generator allows a user to name the nonterminals and tokens used. However, the user has no say in the name of the nonterminals in the generated AST grammar. The AST grammar is generated

following the rules described earlier, so although it is consistent, the user cannot change it other than by changing the original nonterminal names. It would be interesting to show the user the proposed AST grammar and let the user change the names. For example, 2 of the alternatives in the Factor nonterminal in the EBNF are called "Factor_integerliteral" and "Factor_stringliteral" in the AST. The user may have preferred names such as "NumFactor" and "StringFactor". The naming doesn't impact the performance or logic of the code but could make the code easier to work with for the user if the names have more meaning than the generated names do. This is especially important because the names of the AST nonterminals are the names of the generated AST Java objects that encapsulate the source program.

Conclusion

The goal of this thesis was to write a parser generator for an LL(1) grammar in EBNF format. 200 pages of code later (see Appendix I) this task has been accomplished. I quickly realized that generating only a parser was not sufficient because a scanner is needed in order to test the parser. I could have manually written a scanner for each parser I generated, but that seemed to defeat the purpose of automating code. Thus, I extended the parser generator to also generate a simple yet flexible scanner.

In order to generate the scanner, I needed to read in the token names and definitions. Because there is no standard way of doing this, I created a proprietary format that I could easily parse. I did not actually write a full-blown scanner and parser for this format because it was easy enough to parse using simpler techniques. Since regular expressions were created to do string matching, it seemed natural to define the token definition using regular expressions instead of using some other notation like EBNF.

Generating the parser required more work than generating the scanner. I first did have to write a scanner and parser for the EBNF input grammar. That grammar was then transformed into a simpler AST grammar. The starter set for each AST production rule was then calculated. Given the AST grammar, a set of algorithms were applied to construct the AST java classes. The parser itself was then generated following a different algorithm using the starter sets that were calculated.

The ability to have scanners and parsers generated automatically can save developers a significant amount of time from writing and troubleshooting their

own code. Since the rules outlined in this paper describe an algorithm that can be followed to the letter to systematically generate a scanner and a parser, it makes little sense to manually write one unless it is for educational purposes or to try an optimize something. As indicated earlier, the scanner generated can be optimized for a particular language. The parser, however, is efficient for any language.

If the language you are working with is a programming language, you will then probably need to make sure that each variable being used is defined and is of the right type. You will then probably convert the source program from the user-entered high-level code into a lower-level code that an interpreter can then execute. If your language is not a programming language and is something like XML, you will do other checks such making sure that the beginning and end tags match. Whatever you choose to do, having a scanner and parser for your language that read a source code and represents it into an AST tree is the first step. Automating this first step with the Parser Generator represents a significant time savings that lets the user focus more on subtler aspects of the language with which they are working.

Bibliography

1) David A Watt & Deryck F Brown
   Programming Language Processors in Java, Compilers and Interpreters
   Pearson Education Limited 2000

2) Joshua Kerievsky
   Refactoring to Patterns
   Pearson Education Inc 2005

3) Terence Parr
   ANTLR Parser Generator and Translator Generator
   http://www.antlr.org/

4) Matt Davis
   Study of scrambling of words
   http://www.mrc-cbu.cam.ac.uk/personal/matt.davis/Cmabrigde/

5) Regular expressions in JDK 1.4.2
   http://java.sun.com/j2se/1.4.2/docs/api/java/util/regex/Pattern.html#sum

6) Jan Goyvaerts
   Regular Expressions Tutorial
   http://www.regular-expressions.info/tutorial.html

7) SML/NJ Fellowship
   Standard ML of New Jersey, a compiler for Standard ML '97
   http://www.smlnj.org/index.html

8) Andrew Cumming
   A Gentle Introduction to ML
   http://www.dcs.napier.ac.uk/course-notes/sml/manual.html

9) The Apache Software Foundation
   The Apache Jakarta Project
   http://jakarta.apache.org

10) XML Tutorial
   W3 Schools
   http://www.w3schools.com/xml/default.asp

11) Whitespace language
   Durham University Computing Society
   http://compsoc.dur.ac.uk/whitespace/

Table of Figures and Tables

Appendix A - EBNF for EBNF Grammar

```
Grammar           ::= ProductionRuleList .
ProductionRuleList::= ProductionRule {ProductionRuleList} .
ProductionRule    ::= NonTerminal "::=" Definition ".".
Definition        ::= DefinitionSequence { "|" DefinitionSequence } .
DefinitionSequence::= DefinitionBody DefinitionSequence .
DefinitionBody    ::= Symbol |
                      "(" Definition ")" |
                      "[" Definition "]" |
                      "{" Definition "}".
Symbol            ::= Terminal | NonTerminal .
NonTerminal       ::= Identifier .
Terminal          ::= """ String """.
```

Appendix B – Starter and Follow Sets for EBNF Grammar

Nonterminal	**Starter Set**
Grammar	Identifier
ProductionRuleList	Identifier
ProductionRule	Identifier
Definition	"(", "[", "{", """, Identifier
DefinitionSequence	"(", "[", "{", """, Identifier
DefinitionBody	"(", "[", "{", """, Identifier
Symbol	""", Identifier
NonTerminal	Identifier
Terminal	"""

Nonterminal	**Follow Set**
Grammar	EOT
ProductionRuleList	EOT, Identifier
ProductionRule	EOT, Identifier
Definition	".", ")", "]", "}"
DefinitionSequence	"|", ".", ")", "]", "}"
DefinitionBody	"(", "[", "{", """, Identifier
Symbol	"(", "[", "{", """, Identifier
NonTerminal	"(", "[", "{", """, Identifier
Terminal	"(", "[", "{", """, Identifier

EBNF is LL(1):

Rule#1: Definition is the only nonterminal with alternatives and the starter set of each alternative is disjoint as can be seen from the table.

Rule #2: We now check the starter set of each grouping that can be empty and its follow set. We have 2 cases in the ProductionRuleList and in the Definition nonterminals.

Groupings that can be empty	**Starter set of grouping**	**Follow set of grouping**
{ProductionRuleList}	Identifier	EOT
{ "|" DefinitionSequence }	"|"	".", ")", "]", "}"

Since the sets are disjoint, rule #2 is satisfied.

Appendix C - AST for EBNF Grammar

```
Grammar                 ::= ProductionRuleList .
ProductionRuleList      ::= ProductionRule ProductionRuleListTail .
ProductionRuleListTail  ::= ProductionRuleList ProductionRuleListTail .
ProductionRule          ::= NonTerminal Definition .
Definition              ::= DefinitionSequence DefinitionTail .
DefinitionTail          ::= DefinitionSequence DefinitionTail .
DefinitionSequence      ::= DefinitionBody DefinitionSequence .
DefinitionBody          ::= SymbolDefinition |
                            GroupingDefinition |
                            ZeroOrOneDefinition |
                            ZeroOrMoreDefinition .
SymbolDefinition        ::= Symbol.
GroupingDefinition      ::= Definition .
ZeroOrOneDefinition     ::= Definition .
ZeroOrMoreDefinition    ::= Definition .
Symbol                  ::= TerminalSymbol |
                            NonTerminalSymbol .
TerminalSymbol          ::= Terminal .
NonTerminalSymbol       ::= NonTerminal.
NonTerminal             ::= Identifier .
Terminal                ::= String .
```

Appendix D - AST for AST Grammar

```
Grammar                 ::= ProductionRuleList .
ProductionRuleList      ::= ProductionRule ProductionRuleListTail .
ProductionRuleListTail  ::= ProductionRule ProductionRuleListTail .
ProductionRule          ::= NonTerminal Definition .
Definition              ::= Symbol AlternativeDefinition SequenceDefinition .
AlternativeDefinition   ::= Symbol AlternativeDefinition .
SequenceDefinition      ::= Symbol SequenceDefinition .
Symbol                  ::= Terminal | NonTerminal .
```

Appendix E - ML EBNF in AST

```
ebnfast.Grammar
 ebnfast.ProductionRuleList
  ebnfast.ProductionRule
   ebnfast.NonTerminal
   ***Program***
   ebnfast.Definition
    ebnfast.DefinitionSequence
     ebnfast.ZeroOrMoreDefinition
      ebnfast.Definition
       ebnfast.DefinitionSequence
        ebnfast.SymbolDefinition
         ebnfast.NonTerminalSymbol
          ebnfast.NonTerminal
          ***Functions***
        ebnfast.DefinitionSequence
         ebnfast.SymbolDefinition
          ebnfast.TerminalSymbol
           ebnfast.Terminal
           ***;***
     ebnfast.DefinitionSequence
      ebnfast.SymbolDefinition
       ebnfast.NonTerminalSymbol
        ebnfast.NonTerminal
        ***Call***
      ebnfast.DefinitionSequence
       ebnfast.SymbolDefinition
        ebnfast.TerminalSymbol
         ebnfast.Terminal
         ***;***
  ebnfast.ProductionRuleListTail
   ebnfast.ProductionRuleList
    ebnfast.ProductionRule
     ebnfast.NonTerminal
     ***Functions***
     ebnfast.Definition
      ebnfast.DefinitionSequence
       ebnfast.SymbolDefinition
        ebnfast.TerminalSymbol
         ebnfast.Terminal
         ***fun***
       ebnfast.DefinitionSequence
        ebnfast.SymbolDefinition
         ebnfast.NonTerminalSymbol
          ebnfast.NonTerminal
          ***Identifier***
        ebnfast.DefinitionSequence
         ebnfast.SymbolDefinition
          ebnfast.TerminalSymbol
           ebnfast.Terminal
           ***(***
         ebnfast.DefinitionSequence
          ebnfast.ZeroOrOneDefinition
           ebnfast.Definition
```

```
        ebnfast.DefinitionSequence
         ebnfast.SymbolDefinition
          ebnfast.NonTerminalSymbol
           ebnfast.NonTerminal
           ***ParList***
       ebnfast.DefinitionSequence
        ebnfast.SymbolDefinition
         ebnfast.TerminalSymbol
          ebnfast.Terminal
          ***)***
        ebnfast.DefinitionSequence
         ebnfast.SymbolDefinition
          ebnfast.TerminalSymbol
           ebnfast.Terminal
           ***:***
         ebnfast.DefinitionSequence
          ebnfast.SymbolDefinition
           ebnfast.NonTerminalSymbol
            ebnfast.NonTerminal
            ***Type***
          ebnfast.DefinitionSequence
           ebnfast.SymbolDefinition
            ebnfast.TerminalSymbol
             ebnfast.Terminal
             ***=***
           ebnfast.DefinitionSequence
            ebnfast.SymbolDefinition
             ebnfast.NonTerminalSymbol
              ebnfast.NonTerminal
              ***Expression***
ebnfast.ProductionRuleListTail
 ebnfast.ProductionRuleList
  ebnfast.ProductionRule
   ebnfast.NonTerminal
   ***ParList***
   ebnfast.Definition
    ebnfast.DefinitionSequence
     ebnfast.SymbolDefinition
      ebnfast.NonTerminalSymbol
       ebnfast.NonTerminal
       ***Par***
     ebnfast.DefinitionSequence
      ebnfast.ZeroOrMoreDefinition
       ebnfast.Definition
        ebnfast.DefinitionSequence
         ebnfast.SymbolDefinition
          ebnfast.TerminalSymbol
           ebnfast.Terminal
           ***,***
         ebnfast.DefinitionSequence
          ebnfast.SymbolDefinition
           ebnfast.NonTerminalSymbol
            ebnfast.NonTerminal
            ***Par***
  ebnfast.ProductionRuleListTail
   ebnfast.ProductionRuleList
    ebnfast.ProductionRule
```

```
 ebnfast.NonTerminal
 ***Par***
 ebnfast.Definition
  ebnfast.DefinitionSequence
   ebnfast.SymbolDefinition
    ebnfast.NonTerminalSymbol
     ebnfast.NonTerminal
     ***Identifier***
   ebnfast.DefinitionSequence
    ebnfast.SymbolDefinition
     ebnfast.TerminalSymbol
      ebnfast.Terminal
      ***:***
    ebnfast.DefinitionSequence
     ebnfast.SymbolDefinition
      ebnfast.NonTerminalSymbol
       ebnfast.NonTerminal
       ***Type***
ebnfast.ProductionRuleListTail
 ebnfast.ProductionRuleList
  ebnfast.ProductionRule
   ebnfast.NonTerminal
   ***Type***
   ebnfast.Definition
    ebnfast.DefinitionSequence
     ebnfast.SymbolDefinition
      ebnfast.TerminalSymbol
       ebnfast.Terminal
       ***int***
    ebnfast.DefinitionTail
     ebnfast.DefinitionSequence
      ebnfast.SymbolDefinition
       ebnfast.TerminalSymbol
        ebnfast.Terminal
        ***real***
     ebnfast.DefinitionTail
      ebnfast.DefinitionSequence
       ebnfast.SymbolDefinition
        ebnfast.TerminalSymbol
         ebnfast.Terminal
         ***string***
      ebnfast.DefinitionTail
       ebnfast.DefinitionSequence
        ebnfast.SymbolDefinition
         ebnfast.TerminalSymbol
          ebnfast.Terminal
          ***list***
  ebnfast.ProductionRuleListTail
   ebnfast.ProductionRuleList
    ebnfast.ProductionRule
     ebnfast.NonTerminal
     ***Expression***
     ebnfast.Definition
      ebnfast.DefinitionSequence
       ebnfast.SymbolDefinition
        ebnfast.NonTerminalSymbol
         ebnfast.NonTerminal
```

```
      ***ConditionalExpression***
   ebnfast.DefinitionTail
    ebnfast.DefinitionSequence
     ebnfast.SymbolDefinition
      ebnfast.NonTerminalSymbol
       ebnfast.NonTerminal
       ***BasicExpression***
ebnfast.ProductionRuleListTail
 ebnfast.ProductionRuleList
  ebnfast.ProductionRule
   ebnfast.NonTerminal
   ***ConditionalExpression***
   ebnfast.Definition
    ebnfast.DefinitionSequence
     ebnfast.SymbolDefinition
      ebnfast.TerminalSymbol
       ebnfast.Terminal
       ***if***
     ebnfast.DefinitionSequence
      ebnfast.SymbolDefinition
       ebnfast.NonTerminalSymbol
        ebnfast.NonTerminal
        ***BasicExpression***
      ebnfast.DefinitionSequence
       ebnfast.SymbolDefinition
        ebnfast.TerminalSymbol
         ebnfast.Terminal
         ***then***
       ebnfast.DefinitionSequence
        ebnfast.SymbolDefinition
         ebnfast.NonTerminalSymbol
          ebnfast.NonTerminal
          ***Expression***
        ebnfast.DefinitionSequence
         ebnfast.SymbolDefinition
          ebnfast.TerminalSymbol
           ebnfast.Terminal
           ***else***
         ebnfast.DefinitionSequence
          ebnfast.SymbolDefinition
           ebnfast.NonTerminalSymbol
            ebnfast.NonTerminal
            ***Expression***
   ebnfast.ProductionRuleListTail
    ebnfast.ProductionRuleList
     ebnfast.ProductionRule
      ebnfast.NonTerminal
      ***BasicExpression***
      ebnfast.Definition
       ebnfast.DefinitionSequence
        ebnfast.SymbolDefinition
         ebnfast.NonTerminalSymbol
          ebnfast.NonTerminal
          ***PrimaryExpression***
        ebnfast.DefinitionSequence
         ebnfast.ZeroOrMoreDefinition
          ebnfast.Definition
```

```
      ebnfast.DefinitionSequence
       ebnfast.SymbolDefinition
        ebnfast.NonTerminalSymbol
         ebnfast.NonTerminal
         ***PrimaryOperator***
       ebnfast.DefinitionSequence
        ebnfast.SymbolDefinition
         ebnfast.NonTerminalSymbol
          ebnfast.NonTerminal
          ***PrimaryExpression***
ebnfast.ProductionRuleListTail
 ebnfast.ProductionRuleList
  ebnfast.ProductionRule
   ebnfast.NonTerminal
   ***PrimaryOperator***
   ebnfast.Definition
    ebnfast.DefinitionSequence
     ebnfast.SymbolDefinition
      ebnfast.TerminalSymbol
       ebnfast.Terminal
       ***or***
    ebnfast.DefinitionTail
     ebnfast.DefinitionSequence
      ebnfast.SymbolDefinition
       ebnfast.TerminalSymbol
        ebnfast.Terminal
        ***and***
  ebnfast.ProductionRuleListTail
   ebnfast.ProductionRuleList
    ebnfast.ProductionRule
     ebnfast.NonTerminal
     ***PrimaryExpression***
     ebnfast.Definition
      ebnfast.DefinitionSequence
       ebnfast.SymbolDefinition
        ebnfast.NonTerminalSymbol
         ebnfast.NonTerminal
         ***SimpleExpression***
       ebnfast.DefinitionSequence
        ebnfast.ZeroOrOneDefinition
         ebnfast.Definition
          ebnfast.DefinitionSequence
           ebnfast.SymbolDefinition
            ebnfast.NonTerminalSymbol
             ebnfast.NonTerminal
             ***RelationalOperator***
           ebnfast.DefinitionSequence
            ebnfast.SymbolDefinition
             ebnfast.NonTerminalSymbol
              ebnfast.NonTerminal
              ***SimpleExpression***
    ebnfast.ProductionRuleListTail
     ebnfast.ProductionRuleList
      ebnfast.ProductionRule
       ebnfast.NonTerminal
       ***RelationalOperator***
       ebnfast.Definition
```

```
  ebnfast.DefinitionSequence
   ebnfast.SymbolDefinition
    ebnfast.TerminalSymbol
     ebnfast.Terminal
     ***<***
   ebnfast.DefinitionTail
    ebnfast.DefinitionSequence
     ebnfast.SymbolDefinition
      ebnfast.TerminalSymbol
       ebnfast.Terminal
       ***<=***
     ebnfast.DefinitionTail
      ebnfast.DefinitionSequence
       ebnfast.SymbolDefinition
        ebnfast.TerminalSymbol
         ebnfast.Terminal
         ***>***
       ebnfast.DefinitionTail
        ebnfast.DefinitionSequence
         ebnfast.SymbolDefinition
          ebnfast.TerminalSymbol
           ebnfast.Terminal
           ***>=***
         ebnfast.DefinitionTail
          ebnfast.DefinitionSequence
           ebnfast.SymbolDefinition
            ebnfast.TerminalSymbol
             ebnfast.Terminal
             ***=***
           ebnfast.DefinitionTail
            ebnfast.DefinitionSequence
             ebnfast.SymbolDefinition
              ebnfast.TerminalSymbol
               ebnfast.Terminal
               ***\=***
ebnfast.ProductionRuleListTail
 ebnfast.ProductionRuleList
  ebnfast.ProductionRule
   ebnfast.NonTerminal
   ***SimpleExpression***
   ebnfast.Definition
    ebnfast.DefinitionSequence
     ebnfast.ZeroOrOneDefinition
      ebnfast.Definition
       ebnfast.DefinitionSequence
        ebnfast.SymbolDefinition
         ebnfast.TerminalSymbol
          ebnfast.Terminal
          ***~***
       ebnfast.DefinitionSequence
        ebnfast.SymbolDefinition
         ebnfast.NonTerminalSymbol
          ebnfast.NonTerminal
          ***Term***
        ebnfast.DefinitionSequence
         ebnfast.ZeroOrMoreDefinition
          ebnfast.Definition
```

```
          ebnfast.DefinitionSequence
           ebnfast.SymbolDefinition
            ebnfast.NonTerminalSymbol
             ebnfast.NonTerminal
             ***AddingOperator***
           ebnfast.DefinitionSequence
            ebnfast.SymbolDefinition
             ebnfast.NonTerminalSymbol
              ebnfast.NonTerminal
              ***Term***
ebnfast.ProductionRuleListTail
 ebnfast.ProductionRuleList
  ebnfast.ProductionRule
   ebnfast.NonTerminal
   ***AddingOperator***
   ebnfast.Definition
    ebnfast.DefinitionSequence
     ebnfast.SymbolDefinition
      ebnfast.TerminalSymbol
       ebnfast.Terminal
       ***+***
    ebnfast.DefinitionTail
     ebnfast.DefinitionSequence
      ebnfast.SymbolDefinition
       ebnfast.TerminalSymbol
        ebnfast.Terminal
        ***-***
  ebnfast.ProductionRuleListTail
   ebnfast.ProductionRuleList
    ebnfast.ProductionRule
     ebnfast.NonTerminal
     ***Term***
     ebnfast.Definition
      ebnfast.DefinitionSequence
       ebnfast.SymbolDefinition
        ebnfast.NonTerminalSymbol
         ebnfast.NonTerminal
         ***Factor***
       ebnfast.DefinitionSequence
        ebnfast.ZeroOrMoreDefinition
         ebnfast.Definition
          ebnfast.DefinitionSequence
           ebnfast.SymbolDefinition
            ebnfast.NonTerminalSymbol
             ebnfast.NonTerminal
             ***MultiplyingOperator***
           ebnfast.DefinitionSequence
            ebnfast.SymbolDefinition
             ebnfast.NonTerminalSymbol
              ebnfast.NonTerminal
              ***Factor***
    ebnfast.ProductionRuleListTail
     ebnfast.ProductionRuleList
      ebnfast.ProductionRule
       ebnfast.NonTerminal
       ***MultiplyingOperator***
       ebnfast.Definition
```

```
      ebnfast.DefinitionSequence
       ebnfast.SymbolDefinition
        ebnfast.TerminalSymbol
         ebnfast.Terminal
         *******
      ebnfast.DefinitionTail
       ebnfast.DefinitionSequence
        ebnfast.SymbolDefinition
         ebnfast.TerminalSymbol
          ebnfast.Terminal
          ***/***
       ebnfast.DefinitionTail
        ebnfast.DefinitionSequence
         ebnfast.SymbolDefinition
          ebnfast.TerminalSymbol
           ebnfast.Terminal
           ***div***
        ebnfast.DefinitionTail
         ebnfast.DefinitionSequence
          ebnfast.SymbolDefinition
           ebnfast.TerminalSymbol
            ebnfast.Terminal
            ***mod***
    ebnfast.ProductionRuleListTail
     ebnfast.ProductionRuleList
      ebnfast.ProductionRule
       ebnfast.NonTerminal
       ***Factor***
       ebnfast.Definition
        ebnfast.DefinitionSequence
         ebnfast.SymbolDefinition
          ebnfast.NonTerminalSymbol
           ebnfast.NonTerminal
           ***IntegerLiteral***
        ebnfast.DefinitionTail
         ebnfast.DefinitionSequence
          ebnfast.SymbolDefinition
           ebnfast.NonTerminalSymbol
            ebnfast.NonTerminal
            ***StringLiteral***
         ebnfast.DefinitionTail
          ebnfast.DefinitionSequence
           ebnfast.SymbolDefinition
            ebnfast.NonTerminalSymbol
             ebnfast.NonTerminal
             ***Identifier***
           ebnfast.DefinitionSequence
            ebnfast.ZeroOrOneDefinition
             ebnfast.Definition
              ebnfast.DefinitionSequence
               ebnfast.SymbolDefinition
                ebnfast.TerminalSymbol
                 ebnfast.Terminal
                 ***(***
               ebnfast.DefinitionSequence
                ebnfast.ZeroOrOneDefinition
                 ebnfast.Definition
```

```
                    ebnfast.DefinitionSequence
                     ebnfast.SymbolDefinition
                      ebnfast.NonTerminalSymbol
                       ebnfast.NonTerminal
                       ***ArgList***
                  ebnfast.DefinitionSequence
                   ebnfast.SymbolDefinition
                    ebnfast.TerminalSymbol
                     ebnfast.Terminal
                     ***)***
              ebnfast.DefinitionTail
               ebnfast.DefinitionSequence
                ebnfast.SymbolDefinition
                 ebnfast.NonTerminalSymbol
                  ebnfast.NonTerminal
                  ***ListLiteral***
               ebnfast.DefinitionTail
                ebnfast.DefinitionSequence
                 ebnfast.SymbolDefinition
                  ebnfast.TerminalSymbol
                   ebnfast.Terminal
                   ***(***
                 ebnfast.DefinitionSequence
                  ebnfast.SymbolDefinition
                   ebnfast.NonTerminalSymbol
                    ebnfast.NonTerminal
                    ***Expression***
                  ebnfast.DefinitionSequence
                   ebnfast.SymbolDefinition
                    ebnfast.TerminalSymbol
                     ebnfast.Terminal
                     ***)***
                ebnfast.DefinitionTail
                 ebnfast.DefinitionSequence
                  ebnfast.SymbolDefinition
                   ebnfast.TerminalSymbol
                    ebnfast.Terminal
                    ***not***
                  ebnfast.DefinitionSequence
                   ebnfast.SymbolDefinition
                    ebnfast.NonTerminalSymbol
                     ebnfast.NonTerminal
                     ***Factor***
        ebnfast.ProductionRuleListTail
         ebnfast.ProductionRuleList
          ebnfast.ProductionRule
           ebnfast.NonTerminal
           ***ListLiteral***
           ebnfast.Definition
            ebnfast.DefinitionSequence
             ebnfast.SymbolDefinition
              ebnfast.TerminalSymbol
               ebnfast.Terminal
               ***[***
             ebnfast.DefinitionSequence
              ebnfast.ZeroOrOneDefinition
               ebnfast.Definition
```

```
          ebnfast.DefinitionSequence
           ebnfast.SymbolDefinition
            ebnfast.NonTerminalSymbol
             ebnfast.NonTerminal
             ***ExpressionList***
         ebnfast.DefinitionSequence
          ebnfast.SymbolDefinition
           ebnfast.TerminalSymbol
            ebnfast.Terminal
            ***]***
    ebnfast.ProductionRuleListTail
     ebnfast.ProductionRuleList
      ebnfast.ProductionRule
       ebnfast.NonTerminal
       ***ExpressionList***
       ebnfast.Definition
        ebnfast.DefinitionSequence
         ebnfast.SymbolDefinition
          ebnfast.NonTerminalSymbol
           ebnfast.NonTerminal
           ***Expression***
         ebnfast.DefinitionSequence
          ebnfast.ZeroOrMoreDefinition
           ebnfast.Definition
            ebnfast.DefinitionSequence
             ebnfast.SymbolDefinition
              ebnfast.TerminalSymbol
               ebnfast.Terminal
               ***,***
             ebnfast.DefinitionSequence
              ebnfast.SymbolDefinition
               ebnfast.NonTerminalSymbol
                ebnfast.NonTerminal
                ***Expression***
      ebnfast.ProductionRuleListTail
       ebnfast.ProductionRuleList
        ebnfast.ProductionRule
         ebnfast.NonTerminal
         ***Call***
         ebnfast.Definition
          ebnfast.DefinitionSequence
           ebnfast.SymbolDefinition
            ebnfast.NonTerminalSymbol
             ebnfast.NonTerminal
             ***Identifier***
           ebnfast.DefinitionSequence
            ebnfast.SymbolDefinition
             ebnfast.TerminalSymbol
              ebnfast.Terminal
              ***(***
            ebnfast.DefinitionSequence
             ebnfast.ZeroOrOneDefinition
              ebnfast.Definition
               ebnfast.DefinitionSequence
                ebnfast.SymbolDefinition
                 ebnfast.NonTerminalSymbol
                  ebnfast.NonTerminal
```

```
                ***ArgList***
            ebnfast.DefinitionSequence
             ebnfast.SymbolDefinition
              ebnfast.TerminalSymbol
               ebnfast.Terminal
               ***)***
       ebnfast.ProductionRuleListTail
        ebnfast.ProductionRuleList
         ebnfast.ProductionRule
          ebnfast.NonTerminal
          ***ArgList***
          ebnfast.Definition
           ebnfast.DefinitionSequence
            ebnfast.SymbolDefinition
             ebnfast.NonTerminalSymbol
              ebnfast.NonTerminal
              ***Expression***
            ebnfast.DefinitionSequence
             ebnfast.ZeroOrMoreDefinition
              ebnfast.Definition
               ebnfast.DefinitionSequence
                ebnfast.SymbolDefinition
                 ebnfast.TerminalSymbol
                  ebnfast.Terminal
                  ***,***
                ebnfast.DefinitionSequence
                 ebnfast.SymbolDefinition
                  ebnfast.NonTerminalSymbol
                   ebnfast.NonTerminal
                   ***Expression***
```

Appendix F.1 - Language: EMT

AST Package Name: emtast
Scanner/Parser Package Name: emtparser

EBNF:

```
Program ::= SingleCommand .
Command ::= SingleCommand {";" SingleCommand}.
SingleCommand ::= Identifier AssignOrCallCommandTail | "if" Expression "then"
SingleCommand "else" SingleCommand | "while" Expression "do" SingleCommand |
"let" Declaration "in" SingleCommand | "begin" Command "end" | "skip".
AssignOrCallCommandTail ::= { "[" Expression "]" } ":=" Expression | "("
[ActualParameterSequence] ")".
Expression ::= PrimaryExpression {PrimaryOperator PrimaryExpression}.
PrimaryOperator ::= "/\" | "\/".
PrimaryExpression ::= SimpleExpression [RelationalOperator SimpleExpression].
RelationalOperator ::= "<" | "<=" | ">" | ">=" | "=" | "\=".
SimpleExpression ::= ["-"] Term {AddingOperator Term}.
AddingOperator ::= "+" | "-".
Term ::= Factor {MultiplyingOperator Factor}.
MultiplyingOperator ::= "*" | "/" | "//".
Factor ::= IntegerLiteral | CharacterLiteral|Vname | "(" Expression ")" | "\"
Factor.
Vname ::= Identifier {"["Expression"]"}.
ActualParameterSequence ::= ActualParameter ["," ActualParameterSequence].
ActualParameter ::= Expression | "var" Vname.
Declaration ::= singleDeclaration {";" singleDeclaration }.
singleDeclaration ::= "const" Identifier "~" Constant |"var" IdentifierList ":"
Typedenoter | "proc" Identifier "("[FormalParameterSequence]")" "~"
SingleCommand.
Constant ::= IntegerLiteral | CharacterLiteral | Identifier.
IdentifierList ::= Identifier {"," Identifier}.
Typedenoter ::= Identifier | "array" Constant "of" Typedenoter.
FormalParameterSequence ::= FormalParameter ["," FormalParameterSequence].
FormalParameter ::= Identifier ":" Typedenoter | "var" Identifier ":"
Typedenoter.
```

Token:

```
#EMT Lexicon
#IGNORED TOKENS
SEPERATOR:Seperator
COMMENT:Comment

#TOKENS

IDENTIFIER:Identifier
INTEGERLITERAL:IntegerLiteral
CHARACTERLITERAL:CharacterLiteral
ARRAY:"array"
BEGIN:"begin"
CONST:"const"
DO:"do"
```

```
ELSE:"else"
END:"end"
IF:"if"
IN:"in"
LET:"let"
OF:"of"
PROC:"proc"
SKIP:"skip"
THEN:"then"
VAR:"var"
WHILE:"while"
COLON:":"
SEMICOLON:";"
COMMA:","
BECOMES:":="
NOT:"\"
LPAREN:"("
RPAREN:")"
LBRACKET:"["
RBRACKET:"]"
AND:"/\"
ASTERISK:"*"
DIV:"/"
EQUAL:"="
GREATER:">"
GEQ:">="
NEQ:"\="
LESS:"<"
LEQ:"<="
MINUS:"-"
MOD:"//"
OR:"\/"
PLUS:"+"
IS:"~"

#VARIABLES
IntegerLiteral:\d+
CharacterLiteral:'\w'
Identifier:[a-zA-Z]([a-zA-Z]|\d)*
Comment:!(\S? ?)*
Seperator:\x20|\t
```

Sample Code:

```
!This is a comment.It continues to the end - of - line.
let
 const m ~8;
 const j ~8
!var n : integer;
!var x, y, z : array 5 of array 6 of integer;
!proc p() ~let var x : char in begin x : = y;p() end;
!proc q(var x : array 5 of Boolean, y : integer) ~skip
in
 begin
 if (- true) then skip else skip;
 if (true) then skip else skip;
```

```
!x[m + 2] : = 2;
!n : = 2 * m * m;
!putint(n);
!f(a[2][3], var a);
 x := x < y * 6
end
```

Appendix F.2 - Language: Pascal

AST Package Name: pascalast
Scanner/Parser Package Name: pascalparser

EBNF:

```
Program ::= "program" ProgramName ";" BlockBody ".".
BlockBody ::= [ ConstantDefinitionPart ] [ TypeDefinitionPart ]     [
VariableDefinitionPart ] { ProcedureDefinition }     CompoundStatement.
ConstantDefinitionPart ::= "const" ConstantDefinition { ConstantDefinition }.
ConstantDefinition ::= Constant "=" Constant ";".
TypeDefinitionPart ::= "type" TypeDefinition { TypeDefinition }.
TypeDefinition ::= TypeName "=" NewType ";".
NewType ::= NewArrayType | NewRecordType.
NewArrayType ::= "array" "[" IndexRange "]" "of" TypeName.
IndexRange ::= Constant ".." Constant.
NewRecordType ::= "record" FieldList "end".
FieldList ::= RecordSection { ";" RecordSection }.
RecordSection ::= FieldName { "," FieldName  } ":" TypeName.
VariableDefinitionPart ::= "var" VariableDefinitionPart { VariableDefinitionPart
}.
VariableDefinition ::= VariableGroup ";".
VariableGroup ::= VariableName  { "," VariableName } ":" TypeName.
ProcedureDefinition ::= "procedure" ProcedureName ProcedureBlock ";".
ProcedureBlock ::= [ "(" FormalParameterList ")" ] ";" BlockBody .
FormalParameterList ::= ParameterDefinition { ";" ParameterDefinition }.
ParameterDefinition ::= ["var"] VariableGroup.
Statement ::= [StatementBody] .
StatementBody ::= AssignmentOrProcedureStatement | IfStatement | WhileStatement
| CompoundStatement.
AssignmentOrProcedureStatement ::= Name AssignmentOrProcedureStatementTail.
AssignmentOrProcedureStatementTail ::= AssignmentStatementTail |
ProcedureStatementTail.
AssignmentStatementTail ::= VariableAccessTail ":=" Expression.
ProcedureStatementTail ::= [ "(" ActualParameterList ")" ].
ActualParameterList ::= ActualParameterList {"," ActualParameter}.
ActualParameter ::= Expression.
IfStatement ::= "if" Expression "then" Statement [ "else" Statement ].
WhileStatement ::= "while" Expression "do" Statement.
CompoundStatement ::= "{" Statement { ";" Statement } "end".
Expression ::= SimpleExpression [ RelationalOperator SimpleExpression ].
RelationalOperator ::= "<" | "=" | "<>" | ">".
SimpleExpression ::= [ SignOperator ] Term { AddingOperator Term }.
SignOperator ::= "+" | "-".
AddingOperator ::= "+" | "-" | "or".
Term ::= Factor { MultiplyOperator Factor }.
MultiplyOperator ::= "*" | "div" | "mod" | "and".
Factor ::= Constant | VariableAccess | "(" Expression ")" | "not" Factor.
VariableAccess ::= VariableName VariableAccessTail.
VariableAccessTail ::= {Selector}.
Selector ::= IndexSelector | FieldSelector.
IndexSelector ::= "[" Expression "]".
FieldSelector ::= "." FieldName.
Constant ::= Numeral | ConstantName.
```

```
ProgramName ::= Identifier.
TypeName ::= Identifier.
FieldName ::= Identifier.
VariableName ::= Identifier.
ProcedureName ::= Identifier.
Name ::= Identifier.
ConstantName ::= ConstantIdentifier.
```

Token:

```
#EMT Lexicon
#IGNORED TOKENS
SEPERATOR:Seperator

#TOKENS

EMPTY:Empty
NUMERAL:Numeral
IDENTIFIER:Identifier
CONSTANTIDENTIFIER:ConstantIdentifier
PROGRAM:"program"
SEMICOLON:";"
PERIOD:"."
CONST:"const"
EQUALS:"="
TYPE:"type"
ARRAY:"array"
LBRACKET:"["
RBRACKET:"]"
OF:"of"
RANGE:".."
RECORD:"record"
END:"end"
COMMA:","
COLON:":"
VAR:"var"
PROCEDURE:"procedure"
LPAREN:"("
RPAREN:")"
BECOMES:":="
IF:"if"
THEN:"then"
ELSE:"else"
WHILE:"while"
DO:"do"
LCURLYBRACKET:"{"
LESSTHAN:"<"
GREATERTHAN:">"
NOTEQUALS:"<>"
PLUS:"+"
MINUS:"-"
OR:"or"
MULT:"*"
DIV:"div"
MOD:"mod"
AND:"and"
```

```
NOT:"not"

#VARIABLES
Identifier:[a-z]([a-zA-Z]|\d)*
ConstantIdentifier:[A-Z]([A-Z]|\d)*
Numeral:\d+
Empty:
Seperator:\x20|\t
```

Sample Code:

```
program josh;
const LALA = BLAH;
{ while LALA < BLAH do b := 2
end.
```

Appendix F.3 - Language: PL

AST Package Name: plast
Scanner/Parser Package Name: plparser

EBNF:

```
Program ::= Block ".".
Block ::= "begin" DefinitionPart StatementPart "end".
DefinitionPart ::= {Definition ";"}.
Definition ::= ConstantDefinition | VariableDefinition | ProcedureDefinition.
ConstantDefinition ::= "const" ConstantName "=" Constant.
VariableDefinition ::= TypeSymbol VariableDefinitionTail.
VariableDefinitionTail ::= VariableList | "array" VariableList "[" Constant "]".
TypeSymbol ::= "integer" | "boolean".
VariableList ::= VariableName {"," VariableName}.
ProcedureDefinition ::= "proc" ProcedureName[ParameterList] Block.
ParameterList ::= "(" Parameter {"," Parameter}")".
Parameter ::= TypeSymbol ParameterTail.
ParameterTail ::= VariableName | "array" VariableName "[" Constant "]".
StatementPart ::= {Statement ";"}.
Statement ::= EmptyStatement | ReadStatement | WriteStatement |
AssignmentStatement | ProcedureStatement | IfStatement | DoStatement.
EmptyStatement ::= "skip".
ReadStatement ::= "read" VariableAccessList.
WriteStatement ::= "write" ExpressionList.
ExpressionList ::= Expression {"," Expression}.
AssignmentStatement :: VariableAccessList ":=" ExpressionList.
ProcedureStatement ::= "call" ProcedureName.
IfStatement ::= "if" GuardedCommandList "fi".
DoStatement ::= "do" GuardedCommandList "od".
GuardedCommandList ::= GuardedCommand {"[]" GuardedCommand}.
GuardedCommand ::= Expression "->" StatementPart.
Expression ::= PrimaryExpression {PrimaryOperator PrimaryExpression}.
PrimaryOperator ::= "&" | "|".
PrimaryExpression ::= SimpleExpression [RelationalOperator SimpleExpression].
RelationalOperator ::= "<" | "=" | ">".
SimpleExpression ::= ["-"] Term {AddingOperator Term}.
AddingOperator ::= "+" | "-".
Term ::= Factor {MultiplyingOperator Factor}.
MultiplyingOperator ::= "*" | "/" | "\".
Factor ::= Constant | VariableAccess | "(" Expression ")" | "~" Factor.
VariableAccess ::= VariableName [IndexSelector].
VariableAccessList ::= VariableAccess {"," VariableAccess}.
IndexSelector ::= "[" Expression "]".
Constant ::= Numeral|BooleanSymbol|ConstantName.
Numeral ::= Digit{Digit}.
BooleanSymbol ::= "false"|"true".
VariableName ::= Name.
ProcedureName ::= Name.
```

Token:

```
#EMT Lexicon
#IGNORED TOKENS
SEPERATOR:Seperator

#TOKENS

DIGIT:Digit
LETTER:Letter
NAME:Name
CONSTANTNAME:ConstantName
INTEGERLITERAL:"integer"
BOOLEANLITERAL:"boolean"
LPAREN:"("
RPAREN:")"
LBRACKET:"["
RBRACKET:"]"
BEGIN:"begin"
END:"end"
SEMICOLON:";"
CONST:"const"
EQUALS:"="
ARRAY:"array"
COMMA:","
PERIOD:"."
PROC:"proc"
SKIP:"skip"
READ:"read"
WRITE:"write"
BECOMES:":="
CALL:"call"
IFBEGIN:"if"
IFEND:"fi"
DOBEGIN:"do"
DOEND:"od"
EMPTYARRAY:"[]"
POINTER:"->"
AND:"&"
OR:"|"
LESSTHAN:"<"
GREATERTHAN:">"
MINUS:"-"
PLUS:"+"
ASTERIX:"*"
DIV:"/"
MOD:"\"
NOT:"~"
FALSEVALUE:"false"
TRUEVALUE:"true"
UNDERSCORE:"_"

#VARIABLES
Name:[a-z]([a-zA-Z]|\d|_)*
ConstantName:[A-Z]([A-Z]|\d|_)*
Digit:\d
Letter:[a-zA-Z]
Seperator:\x20|\t
```

Sample Code:

```
begin
  const N = 10;
  integer array a[N];
  integer x, i;

  proc search(integer array a[N])
  begin
    integer i, m;
    boolean found;

    i, m := 0, n;
    found := false;
    do
      i < m ->
        if
          a[i] = x -> m := i; found := true;[]
          ~(a[i] = x) -> i := i + 1;
        fi;
    od;
  end;

  i := 0;
  do
    ~(i > n - 1) -> read a[i]; i := i + 1;
  od;
  read x;
  do
    ~(x = 0) ->
      call search;
      if
        found -> write x, i + 1; []
        ~found -> write x;
      fi;
    read x;
  od;
end.
```

Appendix F.4 - Language: Badii

AST Package Name: badiiast
Scanner/Parser Package Name: badiiparser

EBNF:

```
Program ::= {"Global" VarDeclaration} FuncDeclaration { FuncDeclaration }.
VarDeclaration ::= Type Varlist.
Varlist ::= Identifier {"," Identifier}.
Type ::= ("Int" |"Line"|"Point"|"Boolean") ["["IntegerLiteral"]"].
FuncDeclaration ::=  Type Identifier "(" [Paramlist] ")" {VarDeclaration}
StmtBlock "End".
Paramlist ::= Parameter {"," Parameter}.
Parameter ::= Type Identifier.
StmtBlock ::= Stmt {Stmt}.
Stmt ::= Expression ";" | Assignstmt ";" | IfBlock | WhileBlock | Forblock  |
ReturnStmt ";" | BuiltinFunc ";"| ReadStmt ";" | WriteStmt ";".
IfBlock ::= "If" Expression "Then" StmtBlock ["else" StmtBlock] "IfEnd".
WhileBlock ::= "While" Expression "Do" StmtBlock "WhileEnd" .
Forblock ::= ForblockDefault | ForblockLine.
ForblockDefault ::= "For" [Assignstmt] ";" Expression ";" [Assignstmt] "Do"
StmtBlock "ForEnd".
ForblockLine ::= "ForLine" Identifier "In" Identifier "Do" StmtBlock "ForEnd".
Assignstmt ::= "Let" Varaccess "<-" Expression.
Varaccess ::= Identifier ["["Expression"]"] | Identifier "."
("x"|"y"|IntegerLiteral) .
ReturnStmt ::= "Return" Expression.
BuiltinFunc ::= ColorFunc | DrawFunc | AppendFunc.
ColorFunc ::= "Color" Expression ";" Expression ";" Expression.
DrawFunc ::= "Draw" Expression.
AppendFunc ::= "Append" Expression "To" Identifier.
ReadStmt ::=  "Read" Varaccess.
WriteStmt ::= "Write" Expression.
Expression  ::= PrimaryExpression [PrimaryOperator PrimaryExpression].
PrimaryOperator ::= "&&"|"||".
PrimaryExpression ::= EqualsEqualsExpression [EqualsEqualsOperator
EqualsEqualsExpression].
EqualsEqualsOperator ::= "==" | "!=".
EqualsEqualsExpression ::= SimpleExpression [RelationalOperator
SimpleExpression].
RelationalOperator ::= "<" | "<=" | ">" | ">=".
SimpleExpression ::= ["-"] Term {AddingOperator Term}.
AddingOperator ::= "+" | "-".
Term ::= Factor {MultiplyingOperator Factor}.
MultiplyingOperator ::= "*" | "/" | "%".
Factor ::= Constant | Varaccess | "(" Expression ")" | Funccall | "!" Factor.
Constant ::= IntegerLiteral | BooleanLiteral | PointLiteral | LineLiteral.
BooleanLiteral ::= "True"|"False".
PointLiteral ::= "<" SimpleExpression "," SimpleExpression ">".
LineLiteral ::= "{" Factor  Factor {Factor}"}".
Funccall ::= "Call" Identifier "(" [Arglist] ")".
Arglist ::= Expression {"," Expression}.
```

Token:

```
#EMT Lexicon
#IGNORED TOKENS
SEPERATOR:Seperator
COMMENT:Comment
#TOKENS
IDENTIFIER:Identifier
INTEGER:IntegerLiteral
GLOBAL:"Global"
COMMA:","
INT:"Int"
LINE:"Line"
POINT:"Point"
BOOLEAN:"Boolean"
LBRACKET:"["
RBRACKET:"]"
LPARAM:"("
RPARAM:")"
END:"End"
COMMAN:","
SEMICOLON:";"
IF:"If"
THEN:"Then"
ELSE:"else"
IFEND:"IfEnd"
WHILE:"While"
DO:"Do"
WHILEEND:"WhileEnd"
FOR:"For"
FOREND:"ForEnd"
FORLINE:"ForLine"
IN:"In"
LET:"Let"
POINTER:"<-"
PERIOD:"."
CALL:"Call"
X:"x"
Y:"y"
RETURN:"Return"
COLOR:"Color"
DRAW:"Draw"
APPEND:"Append"
TO:"To"
READ:"Read"
WRITE:"Write"
AND:"&&"
OR:"||"
EQUALS:"=="
NOTEQUALS:"!="
LESSTHAN:"<"
LESSTHANEQUALS:"<="
GREATERTHAN:">"
GREATERTHANEQUALS:">="
MINUS:"-"
PLUS:"+"
MULT:"*"
```

```
DIV:"/"
MOD:"%"
NOT:"!"
TRUE:"True"
FALSE:"False"
LCURLY:"{"
RCURLY:"}"
#VARIABLES
Identifier:[a-z]([a-zA-Z]|\d)*
IntegerLiteral:\d+
Seperator:\x20|\t
Comment:\x23([a-zA-Z]|\d|\x20)*
```

Sample Code:

```
Global Int i
Line l() 30==20; End
```

Appendix F.5 - Language: ML

AST Package Name: mlast
Scanner/Parser Package Name: mlparser

EBNF:

```
Program ::= { Functions ";" } Call ";".
Functions ::= "fun" Identifier "(" [ParList] ")" ":" Type "=" Expression.
ParList ::= Par { "," Par }.
Par ::= Identifier ":" Type.
Type ::= SimpleType ["list"].
SimpleType ::= "int" | "real" | "string".
Expression ::= ConditionalExpression | BasicExpression.
ConditionalExpression ::= "if" BasicExpression "then" Expression "else"
Expression.
BasicExpression ::= PrimaryExpression {PrimaryOperator PrimaryExpression}.
PrimaryOperator ::= "or" | "and".
PrimaryExpression ::= SimpleExpression [RelationalOperator SimpleExpression].
RelationalOperator ::= "<" | "<=" | ">" | ">=" | "=" | "\=".
SimpleExpression ::= ["~"] Term {AddingOperator Term}.
AddingOperator ::= "+" | "-".
Term ::= Factor {MultiplyingOperator Factor}.
MultiplyingOperator ::= "*" | "/" | "div" | "mod".
Factor ::= IntegerLiteral | StringLiteral | Identifier ["(" [ArgList] ")"] |
ListLiteral | "(" Expression ")" | "not" Factor.
ListLiteral ::= "[" [ExpressionList] "]".
ExpressionList ::= Expression { "," Expression }.
Call::= Identifier "(" [ArgList] ")".
ArgList ::= Expression {"," Expression }.
```

Token:

```
#EMT Lexicon
#IGNORED TOKENS
SEPERATOR:Seperator

#TOKENS
IDENTIFIER:Identifier
INTEGERLITERAL:IntegerLiteral
STRINGLITERAL:StringLiteral
FUN:"fun"
INT:"int"
REAL:"real"
STRING:"string"
LIST:"list"
ELSE:"else"
IF:"if"
THEN:"then"
COLON:":"
SEMICOLON:";"
COMMA:","
NEG:"~"
NOT:"not"
```

```
LPAREN:"("
RPAREN:")"
LBRACKET:"["
RBRACKET:"]"
AND:"and"
ASTERISK:"*"
DIV:"/"
MOD:"mod"
INTDIV:"div"
EQUAL:"="
GREATER:">"
GEQ:">="
NEQ:"\="
LESS:"<"
LEQ:"<="
MINUS:"-"
OR:"or"
PLUS:"+"

#VARIABLES
IntegerLiteral:\d+
StringLiteral:'\w'
Identifier:[a-zA-Z]([a-zA-Z]|\d)*
Seperator:\x20|\t
```

Sample Code:

```
fun f(n:int):int = if n=0 then 1 else n*f(n-1);
f(4);
```

Appendix F.6 - Language: XML

AST Package Name: xmlast
Scanner/Parser Package Name: xmlparser

EBNF:

```
Program ::= Element.
Element ::= "<" Word ">" [Data] {Element} [Data] "</" Word ">".
Data::= Word {" " Word}.
```

Token:

```
#EMT Lexicon
#IGNORED TOKENS
SEPERATOR:Seperator

#TOKENS
WORD:Word
ELEMENTSTART:"<"
ENDELEMENTSTART:"</"
ELEMENTEND:">"
SPACE:" "

#VARIABLES
Word:[a-zA-Z ]([a-zA-Z ]|\d)*
Seperator:\x20|\t
```

Sample Code:

```
<person>
<firstname>Josh</firstname>
<lastname>Foure</lastname>
</person>
```

Appendix G – Performance Code

In order to test the performance of Sun's DOM parser and my own generated one, I wrote a program called XMLGenerator that generated 2 different kinds of XML files with 1, 10, 100, 1000 and 10000 entries. I then wrote a program called DOMParser that reads each file and outputs the time it took to parse the XML for each case. Note that I read the first 1 entry file 2 times because the first time the program is run it needs to load certain classes in memory and I didn't want to measure that time. In order to test the parser code I generated, I made a few modifications to the Main.java class that are very similar to the DOMParser's main method.

thesis.XMLGenerator.java:

```
package thesis;

import java.util.*;
import java.io.*;

public class XMLGenerator{

    public static void main(String args[]){
        printFile(generateSiblings(1),"siblings1.xml");
        printFile(generateSiblings(10),"siblings10.xml");
        printFile(generateSiblings(100),"siblings100.xml");
        printFile(generateSiblings(1000),"siblings1000.xml");
        printFile(generateSiblings(10000),"siblings10000.xml");
        printFile(generateTree(1),"tree1.xml");
        printFile(generateTree(10),"tree10.xml");
        printFile(generateTree(100),"tree100.xml");
        printFile(generateTree(1000),"tree1000.xml");
        printFile(generateTree(10000),"tree10000.xml");
    }

    public static void printFile(StringBuffer sb, String filename){
        try{
            PrintWriter out = new PrintWriter(new
FileWriter("c:\\dev\\code\\thesis\\"+filename));
            out.print(sb.toString());
            out.flush();
            out.close();
        } catch (Exception e){
            e.printStackTrace();
        }

    }

    public static StringBuffer generateSiblings(int count){
        StringBuffer out = new StringBuffer();
        out.append("<personlist>\n");
        for (int i=0;i<count;i++){
            out.append("<person>\n");
            out.append("<firstname>Josh"+i+"</firstname>\n");
            out.append("<lastname>Foure"+i+"</lastname>\n");
```

```
            out.append("</person>\n");
        }
        out.append("</personlist>\n");
        return out;
    }

    public static StringBuffer generateTree(int count){
        StringBuffer out = new StringBuffer();
        out.append("<personlist>");
        generateTreeChild(out, count);
        out.append("</personlist>");
        return out;
    }

    public static void generateTreeChild(StringBuffer out, int count){
        for (int i=0;i<count;i++){
            out.append("<person>\n");
            out.append("<firstname>Josh"+i+"</firstname>\n");
            out.append("<lastname>Foure"+i+"</lastname>\n");
        }
        for (int i=0;i<count;i++){
            out.append("</person>\n");
        }
    }
}
```

thesis.DOMParser.java:

```
package thesis;

import org.xml.sax.*;
import org.xml.sax.helpers.*;
import org.w3c.dom.*;
import javax.xml.parsers.*;

import java.io.*;
import java.util.*;
/**
 *
 * @author  jfoure
 */
public class DOMParser {

    /** Creates a new instance of DOMParser */
    public DOMParser() {
    }

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        parse("c:\\dev\\code\\thesis\\siblings1.xml");
        parse("c:\\dev\\code\\thesis\\siblings1.xml");
        parse("c:\\dev\\code\\thesis\\siblings10.xml");
        parse("c:\\dev\\code\\thesis\\siblings100.xml");
```

```
        parse("c:\\dev\\code\\thesis\\siblings1000.xml");
        parse("c:\\dev\\code\\thesis\\siblings10000.xml");
        parse("c:\\dev\\code\\thesis\\tree1.xml");
        parse("c:\\dev\\code\\thesis\\tree10.xml");
        parse("c:\\dev\\code\\thesis\\tree100.xml");
        parse("c:\\dev\\code\\thesis\\tree1000.xml");
        parse("c:\\dev\\code\\thesis\\tree10000.xml");
    }

    private static void print(Node node) {
        // Check for children
        if(node.getChildNodes().getLength() > 0) {
            // Output Parent Name
            System.out.println("Parent: " + node.getNodeName());

            NodeList children = node.getChildNodes();

            for(int i = 0; i < node.getChildNodes().getLength(); i++) {

                // Child info
                if(!children.item(i).getNodeName().toString().equals("#text"))
                    System.out.println("Child " + i + ": " +
children.item(i).getNodeName());
            }

            for(int i = 0; i < node.getChildNodes().getLength(); i++) {
                print(children.item(i));
            }
        }
    }


    private static void parse(String filename) {
        try {
            // Build Document from XML
            long time1 = System.currentTimeMillis();
            DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
            DocumentBuilder db = dbf.newDocumentBuilder();
            Document doc = db.parse(filename);
            long time2 = System.currentTimeMillis();
            System.out.println("Total time: "+(time2-time1));
            //print(doc);

        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }

}
```

Appendix H – Parser Generator Packages

application.*: This contains the standalone client code that an end user will run. The Main.java must be executed to start the program.

ast2parser.*: These classes take the internal AST grammar representation and generate the parser among other things.

astast.*: These classes are the classes that the program uses to represent the grammar after it is converted from the user-entered EBNF to a simpler computer-friendly format.

astgenerator.*: This package is not used in the final product. It was used while I was writing the program to easily generate AST classes.

astparser.*: This contains a parser that can read directly parse an AST grammar input. This is not actually used in the end program because the user enters the EBNF grammar that is automatically converted to an AST grammar in memory.

astscanner.*: This contains a scanner that can read directly scan an AST grammar input. This is not actually used in the end program (see astparser.*).

astsets.*: This package contains helper classes to determine starter sets of AST grammars.

common.*: This package contains files that are used by various other components. For example, the TokenI.java is the interface that any Token.java that the program generates will implement.

defaults.*: These classes contain data for predefined grammars. Each class contains data about a single grammar that includes the EBNF for that grammar, the lexicon, a sample test program, a default package name for both the scanner/parser and for the ast objects.

ebnf2ast.*: This package contains the code to convert a user entered EBNF grammar to a simpler AST grammar.

ebnf2parser.*: This package contains the code that generates the Parser.java that is returned to the user based on the simpler AST grammar.

ebnfast.*: These AST objects are used to represent the EBNF grammar that the user enters in memory.

ebnfparser.*: This package contains the parser for the user entered EBNF grammer.

ebnfscanner.*: This package contains the scanner for the user entered EBNF grammer.

ebnfsets.*: This package contains helper classes to determine starter sets of EBNF grammars.

ebnftokengenerator.*: This package contains the code that generates the Scanner.java and Token.java that is returned to the user.

Appendix I – Parser Generator Code

application.BadiiConfiguration.java

```
/*
 * Created on Sep 19, 2004
 */
package application;
import defaults.*;
/**
 * @author jfoure
 */
public class BadiiConfiguration extends ConfigurationImpl{
      public BadiiConfiguration(){
            super(new BadiiSettings(), "c:\\badii");
      }
}
```

application.Configuration.java

```
/*
 * Created on Sep 19, 2004
 */
package application;
import defaults.*;
/**
 * @author jfoure
 */
public interface Configuration extends java.io.Serializable{
      public void setGrammarSettings(GrammarSettings grammarSettings);
      public GrammarSettings getGrammarSettings();
      public void setOutputDirectory(String directory); public String
getOutputDirectory();
}
```

application.ConfigurationImpl.java

```
/*
 * Created on Sep 19, 2004
 */
package application;
import defaults.*;
/**
 * @author jfoure
 */
public class ConfigurationImpl implements Configuration{
      protected GrammarSettings grammarSettings = null;
      protected String directory = null;
      public ConfigurationImpl(GrammarSettings grammarSettings, String
directory){
```

```
            this.grammarSettings = grammarSettings;
            this.directory = directory;
      }

      public void setGrammarSettings(GrammarSettings grammarSettings){
            this.grammarSettings = grammarSettings;
      }
      public GrammarSettings getGrammarSettings(){
            return grammarSettings;
      }


      public void setOutputDirectory(String directory){
            this.directory = directory;
      }
      public String getOutputDirectory(){
            return directory;
      }


}
```

application.EMTConfiguration.java

```
/*
 * Created on Sep 19, 2004
 */
package application;
import defaults.*;
/**
 * @author jfoure
 */
public class EMTConfiguration extends ConfigurationImpl{
      public EMTConfiguration(){
            super(new EMTSettings(), "c:\\emt");
      }
}
```

application.Main.java

```
/*
 * Created on Sep 13, 2004
 */
package application;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;

import defaults.*;
//for action
import astast.Grammar;
```

```
import astast.LineDisplay;
import astast.Visitor;
import ebnfscanner.*;
import ebnfparser.*;
import ebnfast.*;
import ebnfsets.*;
import ebnf2ast.*;
import ast2parser.*;
import ebnftokengenerator.*;
import common.*;
import org.apache.log4j.*;
import java.util.zip.*;

import java.util.*;
/**
 * @author jfoure
 */
public class Main {
    JFrame frame = null;
    JButton submitButton = null;

    FileLoaderPanel grammarFLP = null;
    FileLoaderPanel tokensFLP = null;
    FileLoaderPanel sourceFLP = null;
    JTextField astPackageTextField = null;
    JTextField parserPackageTextField = null;

      JButton outputButton = null;
      JTextField outputTextField = null;
    JTextArea messageTA = new JTextArea();
      JFileChooser outputFC = null;
      JFileChooser configFC = null;

    /**
     * Create the GUI and show it.  For thread safety,
     * this method should be invoked from the
     * event-dispatching thread.
     */
    private void createAndShowGUI() {
        //Make sure we have nice window decorations.
        JFrame.setDefaultLookAndFeelDecorated(true);
        //Create and set up the window.  frame = new JFrame("Josh Foure
Thesis"); frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JPanel astPackagePanel = new JPanel();
        JLabel astPackageLabel = new JLabel("ast package");
        astPackageTextField = new JTextField(30);
        astPackagePanel.add(astPackageLabel);
        astPackagePanel.add(astPackageTextField);
        JPanel parserPackagePanel = new JPanel();
        JLabel parserPackageLabel = new JLabel("parser package");

        parserPackageTextField = new JTextField(30);
        parserPackagePanel.add(parserPackageLabel);
        parserPackagePanel.add(parserPackageTextField);
        submitButton = new JButton("Submit"); submitButton.addActionListener(new
SubmitAction());
```

```
        grammarFLP = new FileLoaderPanel(frame, "EBNF
Grammar","application/josh.txt");
        tokensFLP = new FileLoaderPanel(frame, "Tokens","application/josh.txt");
        sourceFLP = new FileLoaderPanel(frame, "Source
Code","application/josh.txt");
            JPanel outputPanel = new JPanel(); outputTextField = new
JTextField(30);
            //outputTextField.
            outputFC = new JFileChooser();
            outputFC.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);
            outputButton = new JButton("Output Directory");
            outputButton.addActionListener(new ActionListener(){
                  public void actionPerformed(ActionEvent e) {
                        int returnVal = outputFC.showOpenDialog(frame);
                        if (returnVal == JFileChooser.APPROVE_OPTION) {
                              File file = outputFC.getSelectedFile();
                              System.out.println(file.getAbsolutePath());
                              outputTextField.setText(file.getAbsolutePath());
                        }
                  }
            });
            outputPanel.add(outputTextField);
            outputPanel.add(outputButton);

            JScrollPane messageScrollPane = new JScrollPane(messageTA);
            messageScrollPane.setPreferredSize(new Dimension(450, 110));
            messageScrollPane.getHorizontalScrollBar().setValue(0);
            messageScrollPane.getVerticalScrollBar().setValue(0);
        frame.getContentPane().setLayout(new BoxLayout(frame.getContentPane(),
BoxLayout.Y_AXIS));
        //Add the buttons and the log to this panel.
        frame.getContentPane().add(grammarFLP);
        frame.getContentPane().add(astPackagePanel);
        frame.getContentPane().add(parserPackagePanel);
        frame.getContentPane().add(tokensFLP);
        frame.getContentPane().add(sourceFLP);
            frame.getContentPane().add(outputPanel);
            frame.getContentPane().add(submitButton);
        frame.getContentPane().add(messageScrollPane);
        JMenuBar menuBar;
        JMenu fileMenu;
        JMenu predefinedMenu;
        JMenuItem menuItem;

        //        Create the menu bar.
        menuBar = new JMenuBar();

        //Build the file menu.
        fileMenu = new JMenu("File");
        fileMenu.setMnemonic(KeyEvent.VK_F);
        menuBar.add(fileMenu);
        //        a group of JMenuItems
            configFC = new JFileChooser();
            configFC.setFileSelectionMode(JFileChooser.FILES_ONLY);
            menuItem = new JMenuItem("Open Configuration", KeyEvent.VK_O);
            menuItem.addActionListener(new ActionListener(){
                  public void actionPerformed(ActionEvent e) {
```

```
                int returnVal = configFC.showOpenDialog(frame);

                if (returnVal == JFileChooser.APPROVE_OPTION) {
                      File file = configFC.getSelectedFile();
                      try{
                            loadConfiguration(file);
                      } catch (Exception ex){
                            messageTA.setText(ex.getMessage());
                      }
                }
          }
    });
    fileMenu.add(menuItem);
    menuItem = new JMenuItem("Save Configuration", KeyEvent.VK_S);
    menuItem.addActionListener(new ActionListener(){
          public void actionPerformed(ActionEvent e) {
                int returnVal = configFC.showOpenDialog(frame);

                if (returnVal == JFileChooser.APPROVE_OPTION) {
                      File file = configFC.getSelectedFile();
                      try{
                            saveConfiguration(file);
                      } catch (Exception ex){
                            messageTA.setText(ex.getMessage());
                      }
                }
          }
    });
    fileMenu.add(menuItem);

//Build the file menu.
predefinedMenu = new JMenu("Predefined");
predefinedMenu.setMnemonic(KeyEvent.VK_P);
menuBar.add(predefinedMenu);
//        a group of JMenuItems
    menuItem = new JMenuItem("EMT", KeyEvent.VK_E);
    menuItem.addActionListener(new ActionListener() {
          public void actionPerformed(ActionEvent e) {
                setConfiguration(new EMTConfiguration());
          }
    });
    predefinedMenu.add(menuItem);
    menuItem = new JMenuItem("PL", KeyEvent.VK_P);
    menuItem.addActionListener(new ActionListener() {
          public void actionPerformed(ActionEvent e) {
                setConfiguration(new PLConfiguration());
          }
    });
    predefinedMenu.add(menuItem);
    menuItem = new JMenuItem("Pascal", KeyEvent.VK_A);
    menuItem.addActionListener(new ActionListener() {
          public void actionPerformed(ActionEvent e) {
                setConfiguration(new PascalConfiguration());
          }
    });
    predefinedMenu.add(menuItem);
    menuItem = new JMenuItem("Badii", KeyEvent.VK_B);
```

```
            menuItem.addActionListener(new ActionListener() {
                  public void actionPerformed(ActionEvent e) {
                        setConfiguration(new BadiiConfiguration());
                  }
            });
            predefinedMenu.add(menuItem);

        frame.setJMenuBar(menuBar);
        //Display the window.
        frame.pack();
        frame.setVisible(true);
    }

      public void setConfiguration(Configuration configuration){
            setGrammarSettings(configuration.getGrammarSettings());
            outputTextField.setText(configuration.getOutputDirectory());
      }

    public void setGrammarSettings(GrammarSettings settings) {
        grammarFLP.textArea.setText(settings.getGrammar());
tokensFLP.textArea.setText(settings.getTokens());
sourceFLP.textArea.setText(settings.getSourceCode());
astPackageTextField.setText(settings.getASTPackage());
parserPackageTextField.setText(settings.getParserPackage());
    }

    public Configuration retrieveConfiguration(){
      GrammarSettingsImpl grammarSettings = new GrammarSettingsImpl();
            grammarSettings.setASTPackage(astPackageTextField.getText());
            grammarSettings.setParserPackage(parserPackageTextField.getText());
            grammarSettings.setGrammar(grammarFLP.textArea.getText());
            grammarSettings.setTokens(tokensFLP.textArea.getText());
            grammarSettings.setSourceCode(sourceFLP.textArea.getText());
            return new
ConfigurationImpl(grammarSettings,outputTextField.getText());

    }

      public void saveConfiguration(File outFile) throws IOException{
            Configuration config = retrieveConfiguration();
            ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream(outFile));
            out.writeObject(config);
            out.flush();
            out.close();
      }
      public void loadConfiguration(File inFile) throws IOException,
ClassNotFoundException{
            ObjectInputStream in = new ObjectInputStream(new
FileInputStream(inFile));
            Configuration config = (Configuration) in.readObject();
            in.close();
            setConfiguration(config);
      }
```

```
    public static void main(String[] args) throws Exception {
        Properties props = new Properties();
        //Set root logger level to DEBUG and its only appender to A1.
        props.put("log4j.rootLogger", "DEBUG,A1");
        //A1 is set to be a ConsoleAppender.
        props.put("log4j.appender.A1", "org.apache.log4j.ConsoleAppender");
        // A1 uses PatternLayout.
        props.put("log4j.appender.A1.layout", "org.apache.log4j.PatternLayout");
        props.put("log4j.appender.A1.layout.ConversionPattern", "=*=>%-4r [%t]
%-5p %c %x - %m%n");
        PropertyConfigurator.configure(props);
        //Schedule a job for the event-dispatching thread:
        //creating and showing this application's GUI.
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                Main m = new Main();
                m.createAndShowGUI();
            }
        });
    }

    class SubmitAction implements ActionListener {
            File baseDir = null;
        public String generateFilenamePath(String packageName) {
            if (packageName == null)
                return "";
            else
                return "/" + packageName.replace('.', '/') + "/";
        }
        public String generateFilenamePath(GeneratedFile astFile) {
            return generateFilenamePath(astFile.getPackageName()) +
astFile.getFilename();
        }

            public void saveFile(String path, String content) throws
IOException{

      System.out.println("saveFile:"+baseDir.getAbsolutePath()+File.separator+pa
th);
                  FileWriter out = new
FileWriter(baseDir.getAbsolutePath()+File.separator+path);
                  out.write(content);
                  out.flush();
                  out.close();
            }


        public void actionPerformed(ActionEvent ae) {
            //Handle open button action.
                  messageTA.setText("Generating files...");
                  messageTA.repaint();
            String grammarText = grammarFLP.textArea.getText();
            String tokensText = tokensFLP.textArea.getText();
            String sourceText = sourceFLP.textArea.getText();
            String astPackageText = astPackageTextField.getText();
            String parserPackageText = parserPackageTextField.getText();
            try {
```

```
                System.out.println("Creating parser");
                Parser parser = new Parser();
                System.out.println("Generating grammar");
                ebnfast.Grammar grammar = parser.parse(grammarText);
                System.out.println("Setting attribute");
                //
      session.setAttribute("EBNFGRAMMAR",grammar);

                        System.out.println("EBNF ast:");
                new Display(grammar);
                        System.out.println("Done EBNF ast");
                        //Get GenericToken
                        GenericToken genericToken =
TokenGenerator.load(tokensText);

      System.out.println("************************************" + tokensText);
                        System.out.println("constantSpellingMap:" +
GenericToken.constantSpellingMap);
                        System.out.println("variableSpellingMap:" +
GenericToken.variableSpellingMap);
                        //
      session.setAttribute("GENERICTOKEN",genericToken);


                        GrammarChecker grammarChecker = new GrammarChecker();
                        grammarChecker.check(grammar,genericToken);

      System.out.println("definedTerminalSet:"+grammarChecker.getDefinedTerminal
Set());

      System.out.println("usedTerminalSet:"+grammarChecker.getUsedTerminalSet())
;

      System.out.println("undefinedTerminalSet:"+grammarChecker.getUndefinedTerm
inalSet());

                        if
(!grammarChecker.getUndefinedTerminalSet().isEmpty()){
                              messageTA.setText("Some used terminals are not
defined.  They must either be defined in the EBNF or as a Token:
"+grammarChecker.getUndefinedTerminalSet());
                              return;
                        }


                        StarterSetChecker check = new StarterSetChecker();
                        check.check(grammar);
                System.out.println("StarterSet of EBNF");
                StarterSetTable starterSetTable = check.findStarterSet();

                        Iterator starterSetListIterator =
starterSetTable.getStarterSetEntryList().iterator();
                        while (starterSetListIterator.hasNext()) {
                              ebnfsets.StarterSetEntry starterSetEntry
=(ebnfsets.StarterSetEntry) starterSetListIterator.next();

      System.out.print(starterSetEntry.getNonTerminalName() + ": ");
```

```
                        Iterator iterator =
starterSetEntry.getStarterSet().getSymbolSet().iterator();
                        while (iterator.hasNext()) {
                              Symbol symbol = (Symbol) iterator.next();
                              if (symbol instanceof NonTerminalSymbol)
                              System.out.print(((NonTerminalSymbol)
symbol).nonTerminal.spelling);
                              if (symbol instanceof TerminalSymbol)
                              System.out.print("\"" + ((TerminalSymbol)
symbol).terminal.spelling + "\"");
                              if (iterator.hasNext())
                              System.out.print(", ");
                        }

                        System.out.println();
                  }


      System.out.println("%%%%%%%%%%%>"+starterSetTable.getUndefinedTerminalNam
es());

      System.out.println("%%%%%%%%%%%>"+starterSetTable.getUndefinedTerminalNam
es(genericToken));
                // session.setAttribute("STARTERSETTABLE",starterSetTable);



                EBNF2ASTConverter ebnf2ASTConverter = new
EBNF2ASTConverter(genericToken);
                ebnf2ASTConverter.check(grammar);
                astast.Grammar astGrammar = ebnf2ASTConverter.getAstGrammar();
                        System.out.println("PPPPPPPPPPPPPPPPP");
                        astast.LineDisplay lineDisplay = new
astast.LineDisplay(astGrammar);
                        System.out.println("PPPPPPPPPPPPPPPPP");

                //
session.setAttribute("ASTGRAMMAR",ebnf2ASTConverter.getAstGrammar());
                astsets.StarterSetChecker starterSetChecker = new
astsets.StarterSetChecker();
                starterSetChecker.check(astGrammar);
                astsets.StarterSetTable astStarterSetTable =
starterSetChecker.findStarterSet();
                //
session.setAttribute("ASTSTARTERSETTABLE",astStarterSetTable);
                ast2parser.NonTerminalChecker nonTerminalChecker = new
ast2parser.NonTerminalChecker();
                nonTerminalChecker.check(astGrammar);
                ast2parser.AST2Parser ast2Parser = new
ast2parser.AST2Parser(astStarterSetTable,nonTerminalChecker.getUndefinedNonTermi
nalSet(),genericToken,parserPackageText,astPackageText);
                ast2Parser.check(astGrammar);


                GeneratedFile parserAstFile = ast2Parser.generateParser();
                // session.setAttribute("PARSER",parserAstFile);
                java.util.List astList = ast2Parser.getAstFileList();
```

```
                // session.setAttribute("ASTLIST",astList);
                GeneratedFile tokenFile = TokenGenerator.generateToken("Token",
parserPackageText, genericToken.getTokenList());
                // session.setAttribute("TOKEN",tokenFile);
                GeneratedFile scannerFile =
TokenGenerator.generateScanner("Scanner", parserPackageText, null,
genericToken.getTokenList());
                // session.setAttribute("SCANNER",scannerFile);
                GeneratedFile errorFile = ast2Parser.generateError();
                // session.setAttribute("ERROR",errorFile);
                GeneratedFile mainFile = ast2Parser.generateMain();
                // session.setAttribute("MAIN",mainFile);
                GeneratedFile sourceFileFile = ast2Parser.generateSourceFile();
                // session.setAttribute("SOURCEFILE",sourceFileFile);
                GeneratedFile compileBatFile = ast2Parser.generateCompileBat();
                // session.setAttribute("COMPILEBAT",compileBatFile);
                GeneratedFile runBatFile = ast2Parser.generateRunBat();
                //
      session.setAttribute("RUNBAT",runBatFile);

                //
      session.setAttribute("SOURCECODE",grammarForm.getSourceCode());


                        baseDir = new File(outputTextField.getText());
                        baseDir.mkdirs();

                        File astDir = new
File(baseDir.getAbsolutePath()+File.separator+astPackageText);
                        astDir.mkdirs();
                        File parserDir = new
File(baseDir.getAbsolutePath()+File.separator+parserPackageText);
                        parserDir.mkdirs();
                        File commonDir = new
File(baseDir.getAbsolutePath()+File.separator+"common");
                        commonDir.mkdirs();
                        File ebnftokengeneratorDir = new
File(baseDir.getAbsolutePath()+File.separator+"ebnftokengenerator");
                        ebnftokengeneratorDir.mkdirs();


                ZipOutputStream zipOut = new ZipOutputStream(new
FileOutputStream("c:\\josh.zip"));
                zipOut.putNextEntry(new
ZipEntry(generateFilenamePath(parserAstFile)));
                zipOut.write(parserAstFile.getContent().getBytes());
                zipOut.closeEntry();

      saveFile(generateFilenamePath(parserAstFile),parserAstFile.getContent());
                zipOut.putNextEntry(new
ZipEntry(generateFilenamePath(tokenFile)));
                zipOut.write(tokenFile.getContent().getBytes());
                zipOut.closeEntry();

      saveFile(generateFilenamePath(tokenFile),tokenFile.getContent());
                zipOut.putNextEntry(new
ZipEntry(generateFilenamePath(scannerFile)));
```

```
                zipOut.write(scannerFile.getContent().getBytes());
                zipOut.closeEntry();

      saveFile(generateFilenamePath(scannerFile),scannerFile.getContent());
                zipOut.putNextEntry(new
ZipEntry(generateFilenamePath(errorFile)));
                zipOut.write(errorFile.getContent().getBytes());
                zipOut.closeEntry();

      saveFile(generateFilenamePath(errorFile),errorFile.getContent());
                zipOut.putNextEntry(new
ZipEntry(generateFilenamePath(mainFile)));
                zipOut.write(mainFile.getContent().getBytes());
                zipOut.closeEntry();

      saveFile(generateFilenamePath(mainFile),mainFile.getContent());
                zipOut.putNextEntry(new
ZipEntry(generateFilenamePath(sourceFileFile)));
                zipOut.write(sourceFileFile.getContent().getBytes());
                zipOut.closeEntry();

      saveFile(generateFilenamePath(sourceFileFile),sourceFileFile.getContent())
;
                zipOut.putNextEntry(new
ZipEntry(generateFilenamePath(runBatFile)));
                zipOut.write(runBatFile.getContent().getBytes());
                zipOut.closeEntry();

      saveFile(generateFilenamePath(runBatFile),runBatFile.getContent());
                zipOut.putNextEntry(new
ZipEntry(generateFilenamePath(compileBatFile)));
                zipOut.write(compileBatFile.getContent().getBytes());
                zipOut.closeEntry();

      saveFile(generateFilenamePath(compileBatFile),compileBatFile.getContent())
;
                zipOut.putNextEntry(new
ZipEntry(generateFilenamePath(parserAstFile.getPackageName()) + "test.txt"));
                zipOut.write(sourceText.getBytes());
                zipOut.closeEntry();

      saveFile(generateFilenamePath(parserAstFile.getPackageName())+"test.txt",n
ew String(sourceText.getBytes()));
                java.util.List entries = new ArrayList();
                Iterator astListIterator = astList.iterator();
                while (astListIterator.hasNext()) {
                    GeneratedFile astFile = (GeneratedFile)
astListIterator.next();
                    if (!entries.contains(generateFilenamePath(astFile))) {
                        entries.add(generateFilenamePath(astFile));
                        zipOut.putNextEntry(new
ZipEntry(generateFilenamePath(astFile)));
                        zipOut.write(astFile.getContent().getBytes());
                        zipOut.closeEntry();

      saveFile(generateFilenamePath(astFile),astFile.getContent());
                    } else {
```

```
                        System.out.println("Duplicate: " +
generateFilenamePath(astFile));
                    }

                }


                ClassLoader classLoader = this.getClass().getClassLoader();
                        FileOutputStream out = null;
                        InputStream in = null;
                        byte buffer[] = new byte[5000];
                        int length = -1;
                /*
                System.out.println("******common.jar:" +
classLoader.getResourceAsStream("common.jar"));
                        FileOutputStream out = new
FileOutputStream(baseDir.getAbsolutePath()+File.separator+"common.jar");
                InputStream in = classLoader.getResourceAsStream("common.jar");
                zipOut.putNextEntry(new ZipEntry("common.jar"));
                byte buffer[] = new byte[5000];
                int length = -1;
                while ((length = in.read(buffer)) != -1){
                                out.write(buffer, 0, length);
                                zipOut.write(buffer, 0, length);
                }
                        out.flush();
                        out.close();
                zipOut.flush();

      //saveFile(generateFilenamePath(parserAstFile),parserAstFile.getContent())
;
*/

                        in =
classLoader.getResourceAsStream("common/Error.java");
                        out = new
FileOutputStream(baseDir.getAbsolutePath()+File.separator+"common"+File.separato
r+"Error.java");
                        zipOut.putNextEntry(new ZipEntry("Error.java"));
                        buffer = new byte[5000];
                        while ((length = in.read(buffer)) != -1){
                              out.write(buffer, 0, length);
                              zipOut.write(buffer, 0, length);
                        }
                        out.flush();
                        out.close();
                        zipOut.flush();
                        in =
classLoader.getResourceAsStream("common/TokenI.java");
                        out = new
FileOutputStream(baseDir.getAbsolutePath()+File.separator+"common"+File.separato
r+"TokenI.java");
                        zipOut.putNextEntry(new ZipEntry("TokenI.java"));
                        buffer = new byte[5000];
                        while ((length = in.read(buffer)) != -1){
                              out.write(buffer, 0, length);
                              zipOut.write(buffer, 0, length);
```

```
                        }
                        out.flush();
                        out.close();
                        zipOut.flush();
                        in =
classLoader.getResourceAsStream("common/TokenRule.java");
                        out = new
FileOutputStream(baseDir.getAbsolutePath()+File.separator+"common"+File.separato
r+"TokenRule.java");
                        zipOut.putNextEntry(new ZipEntry("TokenRule.java"));
                        buffer = new byte[5000];
                        while ((length = in.read(buffer)) != -1){
                              out.write(buffer, 0, length);
                              zipOut.write(buffer, 0, length);
                        }
                        out.flush();
                        out.close();
                        zipOut.flush();
                        in =
classLoader.getResourceAsStream("common/GenericToken.java");
                        out = new
FileOutputStream(baseDir.getAbsolutePath()+File.separator+"common"+File.separato
r+"GenericToken.java");
                        zipOut.putNextEntry(new ZipEntry("GenericToken.java"));
                        buffer = new byte[5000];
                        while ((length = in.read(buffer)) != -1){
                              out.write(buffer, 0, length);
                              zipOut.write(buffer, 0, length);
                        }
                        out.flush();
                        out.close();
                        zipOut.flush();

                        in =
classLoader.getResourceAsStream("ebnftokengenerator/TokenEntry.java");
                        out = new
FileOutputStream(baseDir.getAbsolutePath()+File.separator+"ebnftokengenerator"+F
ile.separator+"TokenEntry.java");
                        zipOut.putNextEntry(new ZipEntry("TokenEntry.java"));
                        buffer = new byte[5000];
                        while ((length = in.read(buffer)) != -1){
                              out.write(buffer, 0, length);
                              zipOut.write(buffer, 0, length);
                        }
                        out.flush();
                        out.close();
                        zipOut.flush();
                        in =
classLoader.getResourceAsStream("ebnftokengenerator/TokenDefinition.java");
                        out = new
FileOutputStream(baseDir.getAbsolutePath()+File.separator+"ebnftokengenerator"+F
ile.separator+"TokenDefinition.java");
                        zipOut.putNextEntry(new
ZipEntry("TokenDefinition.java"));
                        buffer = new byte[5000];
                        while ((length = in.read(buffer)) != -1){
                              out.write(buffer, 0, length);
```

```
                              zipOut.write(buffer, 0, length);
                        }
                        out.flush();
                        out.close();
                        zipOut.flush();
                        in =
classLoader.getResourceAsStream("ebnftokengenerator/TokenLiteralDefinition.java"
);
                        out = new
FileOutputStream(baseDir.getAbsolutePath()+File.separator+"ebnftokengenerator"+F
ile.separator+"TokenLiteralDefinition.java");
                        zipOut.putNextEntry(new
ZipEntry("TokenLiteralDefinition.java"));
                        buffer = new byte[5000];
                        while ((length = in.read(buffer)) != -1){
                              out.write(buffer, 0, length);
                              zipOut.write(buffer, 0, length);
                        }
                        out.flush();
                        out.close();
                        zipOut.flush();
                        in =
classLoader.getResourceAsStream("ebnftokengenerator/TokenVariableDefinition.java
");
                        out = new
FileOutputStream(baseDir.getAbsolutePath()+File.separator+"ebnftokengenerator"+F
ile.separator+"TokenVariableDefinition.java");
                        zipOut.putNextEntry(new
ZipEntry("TokenVariableDefinition.java"));
                        buffer = new byte[5000];
                        while ((length = in.read(buffer)) != -1){
                              out.write(buffer, 0, length);
                              zipOut.write(buffer, 0, length);
                        }
                        out.flush();
                        out.close();
                        zipOut.flush();




/*
                System.out.println("******log4j1.2.jar");
                in = classLoader.getResourceAsStream("log4j-1.2rc1.jar");
                        out = new
FileOutputStream(baseDir.getAbsolutePath()+File.separator+"log4j-1.2rc1.jar");
                zipOut.putNextEntry(new ZipEntry("log4j-1.2rc1.jar"));
                buffer = new byte[5000];
                while ((length = in.read(buffer)) != -1){
                              out.write(buffer, 0, length);
                    zipOut.write(buffer, 0, length);
                        }
                        out.flush();
                        out.close();
                zipOut.flush();
*/
```

```
      //saveFile(generateFilenamePath(parserAstFile),parserAstFile.getContent())
;
                System.out.println("******all done");
                zipOut.flush();
                zipOut.close();


            } catch (Exception e) {
                System.out.println("Error:" + e);
                        messageTA.setText(e.toString());
                        StackTraceElement ste[] = e.getStackTrace();
                        for (int i=0;i<ste.length;i++){
                              messageTA.append(ste[i].toString()+"\n");
                        }
                        e.printStackTrace();

            }

            messageTA.setText("All done");
        }
    }


    public static StringBuffer loadFile(String filename){
            ClassLoader classLoader = Main.class.getClassLoader();
            StringBuffer out = new StringBuffer();
            try{
                  DataInputStream in = new
DataInputStream(classLoader.getResourceAsStream(filename));
                  String line = null;
                  while ((line=in.readLine()) != null){
                        out.append(line);
                        out.append('\n');
                  }
                  in.close();
            } catch (Exception e){
                  e.printStackTrace();
            }

            return out;
    }

}

class FileLoaderPanel extends JPanel {
    //final JFrame frame = null;
    JLabel label = null;
    JButton button = null;
    JFileChooser fc = null;
    JTextArea textArea = null;
      JButton helpButton = null;
      String helpFilename = null;

    public FileLoaderPanel(final JFrame frame, String labelText, final String
helpFilename) {
        super();
```

```
//        this.frame = mainFrame;
            this.helpFilename = helpFilename;

        label = new JLabel(labelText);

        //Create a file chooser
        fc = new JFileChooser();
        //fc.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);
        fc.setFileSelectionMode(JFileChooser.FILES_ONLY);
        //fc.setFileSelectionMode(JFileChooser.FILES_AND_DIRECTORIES);

        //              In a container that uses a BorderLayout:
        textArea = new JTextArea(5, 30);
        JScrollPane scrollPane = new JScrollPane(textArea);
        scrollPane.setPreferredSize(new Dimension(450, 110));
        //grammarScrollPane.
        //Create the open button.  We use the image from the JLF //Graphics
Repository (but we extracted it from the jar).  button = new JButton("Open " +
labelText + " File...");
        button.addActionListener(new FCFileLoader(frame, button, fc, scrollPane,
textArea));
            helpButton = new JButton("Help");
            helpButton.addActionListener(new ActionListener(){
                  public void actionPerformed(ActionEvent e) {
                        JDialog dialog = new JDialog(frame,true);
                        JTextArea helpText = new JTextArea(5,30);

      helpText.setText(Main.loadFile(helpFilename).toString());
                        JScrollPane helpScrollPane = new JScrollPane(helpText);
                        helpScrollPane.setPreferredSize(new Dimension(450,
110));
                        dialog.getContentPane().add(helpScrollPane);
                        dialog.setSize(450,110);
                        dialog.show();

                  }
            });


        add(label);
        add(scrollPane, BorderLayout.CENTER);
        add(button);
        add(helpButton);
    }

}

class FCFileLoader implements ActionListener {
    JFrame frame = null;
    JButton button = null;
    JFileChooser fc = null;
    JTextArea textArea = null;
    JScrollPane scrollPane = null;

    public FCFileLoader(JFrame frame, JButton button, JFileChooser fc,
JScrollPane scrollPane, JTextArea textArea) {
        this.frame = frame;
```

```
        this.button = button;
        this.fc = fc;
        this.scrollPane = scrollPane;
        this.textArea = textArea;
    }

    public static void readFile(JTextArea textArea, File file) throws Exception
{
        BufferedReader in = new BufferedReader(new FileReader(file));
        String line = null;
        textArea.setText("");
        while ((line = in.readLine()) != null) {
            textArea.append(line);
            textArea.append("\n");
        }

    }
    public void actionPerformed(ActionEvent e) {
        //Handle open button action.
        if (e.getSource() == button) {
            int returnVal = fc.showOpenDialog(frame);
            if (returnVal == JFileChooser.APPROVE_OPTION) {
                File file = fc.getSelectedFile();
                try {
                    readFile(textArea, file);
                } catch (Exception ex) {
                    ex.printStackTrace();
                }


//scrollPane.setCorner(JScrollPane.UPPER_LEFT_CORNER,scrollPane);
                scrollPane.getHorizontalScrollBar().setValue(0);
                scrollPane.getVerticalScrollBar().setValue(0);
                //This is where a real application would open the file.
                System.out.println("Opening: " + file.getName() + ".");
            } else {
                System.out.println("Open command cancelled by user.");
            }
        }
    }
}
```

application.PascalConfiguration.java

```
/*
 * Created on Sep 19, 2004
 */
package application;
import defaults.*;
/**
 * @author jfoure
 */
public class PascalConfiguration extends ConfigurationImpl{
      public PascalConfiguration(){
            super(new PascalSettings(), "c:\\pascal");
```

```
	}
}
```

application.PLConfiguration.java

```
/*
 * Created on Sep 19, 2004
 */
package application;
import defaults.*;
/**
 * @author jfoure
 */
public class PLConfiguration extends ConfigurationImpl{
	public PLConfiguration(){
		super(new PLSettings(), "c:\\pl");
	}
}
```

ast2parser.AST2Parser.java

```
package ast2parser;
import ebnfscanner.*;
import ebnfparser.*;
import astast.*;
import astsets.*;
import common.*;

import java.util.*;
import java.io.*;

public class AST2Parser implements Visitor{
	StarterSetTable starterSetTable = null;
	Set undefinedNonTerminalSet = null;
	TokenI token = null;
	StringBuffer out = new StringBuffer();

	//for ast generation
	private String parserPackageName;
	private String astPackageName;
	private Map alternativeMap = new Hashtable();
	private List allSymbols = new ArrayList();
	private Collection definedNonTerminals = new ArrayList();
	private Collection nonDefinedTerminals = new ArrayList();
	private Collection definedNonTerminalsHelpers = new ArrayList();
	//list of astfiles
	List astFileList = new ArrayList();


	public AST2Parser(StarterSetTable starterSetTable,Set
undefinedNonTerminalSet, TokenI token, String parserPackageName,String
astPackageName){
		this.starterSetTable = starterSetTable;
```

```
            this.token = token;
            this.undefinedNonTerminalSet = undefinedNonTerminalSet;
            this.parserPackageName = parserPackageName;
            this.astPackageName = astPackageName;
      }



      public void check(Grammar grammar){
            astFileList.add(generateAst());
            grammar.visit(this, null);
            astFileList.add(generateVisitor());
astFileList.addAll(generateUndefinedNonTerminals());
astFileList.add(generateDisplay());


      }
      public Object visitGrammar(Grammar grammar, Object arg, int line){
            grammar.productionRuleList.visit(this,null); return null;
      }
      public Object visitProductionRuleList(ProductionRuleList
productionrulelist, Object arg, int line){
            productionrulelist.productionRule.visit(this,null); if
(productionrulelist.productionRuleListTail!=null)
                  productionrulelist.productionRuleListTail.visit(this,null);
            return null;
      }
      public Object visitProductionRuleListTail(ProductionRuleListTail
productionrulelisttail, Object arg, int line){
            productionrulelisttail.productionRule.visit(this,null); if
(productionrulelisttail.productionRuleListTail!=null)

      productionrulelisttail.productionRuleListTail.visit(this,null);
            return null;
      }
      public Object visitProductionRule(ProductionRule productionrule, Object
arg, int line){
            out.append("//ProductionRule:"+productionrule+"\n");
            out.append("private "+productionrule.nonTerminal.spelling+"
parse"+productionrule.nonTerminal.spelling+"(){\n");
//
      out.append("\tlogger.debug(\"parse"+productionrule.nonTerminal.spelling+"(
)\");\n");

            //productionrule.nonTerminal.visit(this,null);

      productionrule.definition.visit(this,productionrule.nonTerminal.spelling);

            out.append("}\n");

            return null;
      }
      public Object visitDefinition(Definition definition, Object arg, int
line){
            String ruleName = (String) arg;
```

```
            //has alternatives
            if (definition.alternativeDefinition!=null){
                  ASTAlternativeHelper alternativeHelper = new
ASTAlternativeHelper(ruleName,isBasicAlternative(definition));
                  alternativeHelper.addAlternative(definition.symbol);
                  out.append("//Check Starter Sets\n"); out.append("\tswitch
(currentToken.kind){\n");
                  StarterSet starterSet =
StarterSetChecker.findStarterSet(definition.symbol,starterSetTable);
                  System.out.println("starterSet:"+starterSet);
                  Iterator starterSetIterator = starterSet.symbolSet.iterator();
                  while (starterSetIterator.hasNext()){
                        Symbol starterSetSymbol =
(Symbol)starterSetIterator.next();
                        if (starterSetSymbol instanceof NonTerminalSymbol){
                              NonTerminalSymbol nonTerminalSymbol =
(NonTerminalSymbol)starterSetSymbol;
                              out.append("\t\tcase
Token."+token.lookupConstantSpelling(nonTerminalSymbol.nonTerminal.spelling)+":\
n");
                        } else {
                              TerminalSymbol terminalSymbol =
(TerminalSymbol)starterSetSymbol;
                              out.append("\t\tcase
Token."+token.lookupConstantSpelling(terminalSymbol.terminal.spelling)+":\n");
                        }
                  }

//out.append("\t\tcase
Token."+token.lookupConstantSpelling(definition.symbol.getSpelling())+":\n");
//out.append("\t\tcase Token.IDENTIFIER:\n");

                  out.append(definition.symbol.getVariableType()+"
"+definition.symbol.getVariableName()+" = null;\n");
                  definition.symbol.visit(this,null);
                  if (alternativeHelper.isBasic())
                        out.append("\t\treturn new
"+alternativeHelper.productionRuleName+"("+definition.symbol.getVariableName()+"
,currentToken.line);");
                  else
                        out.append("\t\treturn
"+definition.symbol.getVariableName()+";");
                  out.append("\n");

      definition.alternativeDefinition.visit(this,alternativeHelper);
                  out.append("\t\tdefault:\n\t\tnew Error(\"Syntax error.
Expected ");
                  Iterator alternativeIterator =
alternativeHelper.alternativeList.iterator();
                  while (alternativeIterator.hasNext()){
                        Symbol symbol = (Symbol) alternativeIterator.next();
                        out.append(symbol.getVariableName());
                        if (alternativeIterator.hasNext())
                              out.append(" or ");
                  }
```

```
                  out.append(".  Got \"+currentToken.getKind()+\" spelled
\"+currentToken.getSpelling()+\".\", currentToken.line);\n\treturn
null;\n\t}\n");
                  GeneratedFile alternativeAstFile =
generateAlternatives(alternativeHelper);
                  astFileList.add(alternativeAstFile);

      System.out.println("###################################"+alternativeAstFile
.getContent());

            }
            else
            {
                  out.append("//parse sequences\n");
                  ASTSequenceHelper sequenceHelper = new
ASTSequenceHelper(ruleName);
                  sequenceHelper.addSequence(definition.symbol);
                  out.append(definition.symbol.getVariableType()+"
"+definition.symbol.getVariableName()+" = null;\n");
                  if (canBeEmpty(definition)){
                        out.append("if (");
                        StarterSet starterSet =
StarterSetChecker.findStarterSet(definition.symbol,starterSetTable);
                        Iterator starterSetIterator =
starterSet.symbolSet.iterator();
                        while (starterSetIterator.hasNext()){
                              Symbol starterSetSymbol =
(Symbol)starterSetIterator.next();
                              if (starterSetSymbol instanceof
NonTerminalSymbol){
                                    NonTerminalSymbol nonTerminalSymbol =
(NonTerminalSymbol)starterSetSymbol;

      out.append("currentToken.kind==Token."+token.lookupConstantSpelling(nonTer
minalSymbol.nonTerminal.spelling));
                              } else {
                                    TerminalSymbol terminalSymbol =
(TerminalSymbol)starterSetSymbol;

      out.append("currentToken.kind==Token."+token.lookupConstantSpelling(termin
alSymbol.terminal.spelling));
                              }

                              if (starterSetIterator.hasNext())
                                    out.append(" || ");
                        }
                        out.append("){\n");
                        definition.symbol.visit(this,null);
                        out.append("}\n");
                  } else {
                        definition.symbol.visit(this,null);
                  }

                  if (definition.sequenceDefinition!=null)

      definition.sequenceDefinition.visit(this,sequenceHelper);
```

```
                  out.append("\t\treturn new
"+sequenceHelper.productionRuleName+"(");
                  List symbolList = sequenceHelper.sequenceList;
                  for (int i=0;i<symbolList.size();i++){
                        Symbol symbol = (Symbol) symbolList.get(i);
                        out.append(symbol.getVariableName()+",");
                  }
                  out.append("currentToken.line);\n\n");
                  GeneratedFile sequenceAstFile =
generateSequences(sequenceHelper);
                  astFileList.add(sequenceAstFile);

      System.out.println("###################################"+sequenceAstFile.ge
tContent());

            }

            return null;
      }
      public Object visitAlternativeDefinition(AlternativeDefinition
alternativedefinition, Object arg, int line){
            ASTAlternativeHelper alternativeHelper = (ASTAlternativeHelper)arg;
            alternativeHelper.addAlternative(alternativedefinition.symbol);
            StarterSet starterSet =
StarterSetChecker.findStarterSet(alternativedefinition.symbol,starterSetTable);
            System.out.println("starterSet:"+starterSet);
            Iterator starterSetIterator = starterSet.symbolSet.iterator();
            while (starterSetIterator.hasNext()){
                  Symbol starterSetSymbol = (Symbol)starterSetIterator.next();
                  if (starterSetSymbol instanceof NonTerminalSymbol){
                        NonTerminalSymbol nonTerminalSymbol =
(NonTerminalSymbol)starterSetSymbol;
                        out.append("\t\tcase
Token."+token.lookupConstantSpelling(nonTerminalSymbol.nonTerminal.spelling)+":\
n");
                  } else {
                        TerminalSymbol terminalSymbol =
(TerminalSymbol)starterSetSymbol;
                        out.append("\t\tcase
Token."+token.lookupConstantSpelling(terminalSymbol.terminal.spelling)+":\n");
                  }
            }

            out.append(alternativedefinition.symbol.getVariableType()+"
"+alternativedefinition.symbol.getVariableName()+" = null;\n");
            alternativedefinition.symbol.visit(this,null);
            if (alternativeHelper.isBasic())
                  out.append("\t\treturn new
"+alternativeHelper.productionRuleName+"("+alternativedefinition.symbol.getVaria
bleName()+",currentToken.line);");
            else
                  out.append("\t\treturn
"+alternativedefinition.symbol.getVariableName()+";");
            out.append("\n");
            if (alternativedefinition.alternativeDefinition!=null)

      alternativedefinition.alternativeDefinition.visit(this,alternativeHelper);
```

```
            return null;
      }
      public Object visitSequenceDefinition(SequenceDefinition
sequencedefinition, Object arg, int line){
            ASTSequenceHelper sequenceHelper = (ASTSequenceHelper)arg;
            sequenceHelper.addSequence(sequencedefinition.symbol);
            out.append(sequencedefinition.symbol.getVariableType()+"
"+sequencedefinition.symbol.getVariableName()+" = null;\n");
            if (canBeEmpty(sequencedefinition)){
                  out.append("if (");
                  StarterSet starterSet =
StarterSetChecker.findStarterSet(sequencedefinition.symbol,starterSetTable);
                  Iterator starterSetIterator = starterSet.symbolSet.iterator();
                  while (starterSetIterator.hasNext()){
                        Symbol starterSetSymbol =
(Symbol)starterSetIterator.next();
                        if (starterSetSymbol instanceof NonTerminalSymbol){
                              NonTerminalSymbol nonTerminalSymbol =
(NonTerminalSymbol)starterSetSymbol;

      out.append("currentToken.kind==Token."+token.lookupConstantSpelling(nonTer
minalSymbol.nonTerminal.spelling));
                        } else {
                              TerminalSymbol terminalSymbol =
(TerminalSymbol)starterSetSymbol;

      out.append("currentToken.kind==Token."+token.lookupConstantSpelling(termin
alSymbol.terminal.spelling));
                        }

                        if (starterSetIterator.hasNext())
                              out.append(" || ");
                  }
                  out.append("){\n");
                  sequencedefinition.symbol.visit(this,null);
                  out.append("}\n");
            } else {
                  sequencedefinition.symbol.visit(this,null);
            }

            if (sequencedefinition.sequenceDefinition!=null)

      sequencedefinition.sequenceDefinition.visit(this,sequenceHelper);
            return null;
      }
      public Object visitTerminalSymbol(TerminalSymbol terminalsymbol, Object
arg, int line){
            out.append(terminalsymbol.variableName+" =
currentToken.spelling;\n");

      terminalsymbol.terminal.visit(this,terminalsymbol.getVariableName());
            return null;
      }
      public Object visitTerminal(Terminal terminal, Object arg, int line){
            String variableName = (String) arg;
```

```
      out.append("accept(Token."+token.lookupConstantSpelling(terminal.spelling)
+");\n");
            return null;
      }
      public Object visitNonTerminalSymbol(NonTerminalSymbol nonterminalsymbol,
Object arg, int line){

      nonterminalsymbol.nonTerminal.visit(this,nonterminalsymbol.getVariableName
());
            return null;
      }
      public Object visitNonTerminal(NonTerminal nonterminal, Object arg, int
line){
            String variableName = (String) arg;
            out.append(variableName+" = parse"+nonterminal.spelling+"();\n");
            return null;
      }


      public boolean canBeEmpty(Definition definition){
            StarterSet starterSet =
starterSetTable.getStarterSet(definition.symbol.getVariableType());
            if (starterSet!=null && starterSet.isCanBeEmpty()==true)
                  return true;
            return definition.symbol.canBeEmpty();
      }
      public boolean canBeEmpty(SequenceDefinition definition){
            StarterSet starterSet =
starterSetTable.getStarterSet(definition.symbol.getVariableType());
            if (starterSet!=null && starterSet.isCanBeEmpty()==true)
                  return true;
            return definition.symbol.canBeEmpty();
      }









      public static void main(String[] args) throws Exception{
            Parser parser = new Parser();
            ebnfast.Grammar ebnfGrammar = parser.parse();
            //new Display(grammar);
            System.out.println("******");

            ebnf2ast.EBNF2ASTConverter ebnf2ASTConverter = new
ebnf2ast.EBNF2ASTConverter(new emt.Token());
            ebnf2ASTConverter.check(ebnfGrammar);
            Grammar astGrammar = ebnf2ASTConverter.getAstGrammar();
            StarterSetChecker starterSetChecker = new StarterSetChecker();
starterSetChecker.check(astGrammar);
```

```
            StarterSetTable starterSetTable =
starterSetChecker.findStarterSet();
            System.out.println("*****");
            NonTerminalChecker nonTerminalChecker = new NonTerminalChecker();
            nonTerminalChecker.check(astGrammar);

      System.out.println("definedNonTerminalSet:"+nonTerminalChecker.getDefinedN
onTerminalSet());

      System.out.println("nonTerminalSet:"+nonTerminalChecker.getUndefinedNonTer
minalSet());

            System.out.println("Generate the parser");
//emt
            AST2Parser ast2Parser = new
AST2Parser(starterSetTable,nonTerminalChecker.getUndefinedNonTerminalSet(),new
emt.Token(),"emtgenerated","emtgenerated.ast");
//minitriangle
//          AST2Parser ast2Parser = new
AST2Parser(starterSetTable,nonTerminalChecker.getUndefinedNonTerminalSet(),new
minitriangle.Token());
            ast2Parser.check(astGrammar);

//emt
            //ebnf2Parser.createParser("C:\\Documents and
Settings\\jfoure\\Desktop\\cs605\\Thesis\\ebnf\\emtgenerated\\Parser.java","emtg
enerated");
            GeneratedFile parserAstFile = ast2Parser.generateParser();
            ast2Parser.writeOutput(parserAstFile);
//          AstFile astAstFile = ast2Parser.generateAst();
//          ast2Parser.writeOutput(astAstFile);

            Iterator astFiles = ast2Parser.astFileList.iterator();
            while (astFiles.hasNext()){
                  GeneratedFile astFile = (GeneratedFile) astFiles.next();
                  ast2Parser.writeOutput(astFile);
            }
/*
            AstFile visitorAstFile = ast2Parser.generateVisitor();
            ast2Parser.writeOutput(visitorAstFile);
            List undefinedNonTerminalList =
ast2Parser.generateUndefinedNonTerminals();
            Iterator undefinedAstFiles = undefinedNonTerminalList.iterator();
            while (undefinedAstFiles.hasNext()){
                  AstFile astFile = (AstFile) undefinedAstFiles.next();
                  ast2Parser.writeOutput(astFile);
            }
*/
//minitriangle
//          ebnf2Parser.createParser("C:\\Documents and
Settings\\jfoure\\Desktop\\cs605\\Thesis\\ebnf\\minitrianglegenerated\\Parser.ja
va","minitrianglegenerated");

            System.out.println(parserAstFile.getContent());
            System.out.println("The program is terminated normally.");

      }
```

```
	/**
	 * Return true if definition only contains single terminal alternatives.
	 * @param definition
	 * @return
	 */
	public static boolean isBasicAlternative(Definition definition){
		//Definition currentDefinition = definition;
		AlternativeDefinition currentAlternativeDefinition =
definition.alternativeDefinition;
		Symbol currentSymbol = definition.symbol;
		do{
			if (!(currentSymbol instanceof TerminalSymbol))
				return false;
			if (currentAlternativeDefinition!=null){
				currentSymbol = currentAlternativeDefinition.symbol;
				currentAlternativeDefinition =
currentAlternativeDefinition.alternativeDefinition;
			}
			else{
				currentSymbol = null;
				currentAlternativeDefinition = null;
			}

		} while (currentSymbol!=null);
		return true;
	}

	public void createParser(String filename,String packageName){
		GeneratedFile parser = generateParser();
		writeOutput(parser);

	}

	public GeneratedFile generateAst(){
		return new GeneratedFile("Ast.java",astPackageName,"package
"+astPackageName+";\n\npublic abstract class Ast{\n\tpublic abstract Object
visit(Visitor v, Object arg);\n}");
	}

	public GeneratedFile generateAlternatives(ASTAlternativeHelper
alternativeHelper){
		definedNonTerminalsHelpers.add(alternativeHelper);
		definedNonTerminals.add(alternativeHelper.productionRuleName);
		if (!alternativeHelper.isBasic()){
			Iterator alternativeIterator =
alternativeHelper.alternativeList.iterator();
			while (alternativeIterator.hasNext()){
				Symbol symbol = (Symbol) alternativeIterator.next();

	alternativeMap.put(symbol.getSpelling(),alternativeHelper.productionRuleNa
me);
			}
			return new
GeneratedFile(alternativeHelper.productionRuleName+".java",astPackageName,"packa
```

```
ge "+astPackageName+";\n\npublic abstract class
"+alternativeHelper.productionRuleName+" extends Ast{}");
            } else {
                  StringBuffer out = new StringBuffer();
                  out.append("package "+astPackageName+";\n\n");
                  out.append("public class ");
                  out.append(alternativeHelper.productionRuleName);
                  out.append(" extends Ast{\n");
                  out.append("public String spelling;\n"); out.append("public
int line;\n");
                  out.append("public
"+alternativeHelper.productionRuleName+"(String Spelling, int line){\n");
                  out.append("  this.spelling = Spelling;\n");
                  out.append("  this.line = line;\n");
                  out.append("}\n");
                  out.append("public Object visit(Visitor v, Object arg){\n");
                  out.append("  return
v.visit"+alternativeHelper.productionRuleName+"(this, arg, line);\n");
                  out.append("}\n");
                  out.append("}");

                  return new
GeneratedFile(alternativeHelper.productionRuleName+".java",astPackageName,out.to
String());

            }
      }

      private GeneratedFile generateSequences(ASTSequenceHelper sequenceHelper){
            definedNonTerminalsHelpers.add(sequenceHelper);
            definedNonTerminals.add(sequenceHelper.productionRuleName);
            if (!allSymbols.contains(sequenceHelper.productionRuleName))
                  allSymbols.add(sequenceHelper.productionRuleName);
            List symbolList = sequenceHelper.sequenceList;
            for (int i=0;i<symbolList.size();i++){
                  Symbol symbol = (Symbol) symbolList.get(i);
                  if (symbol instanceof NonTerminalSymbol &&
!allSymbols.contains(symbol.getSpelling()))
                        allSymbols.add(symbol.getSpelling());
            }
            /*
                        StringTokenizer st = new StringTokenizer(rule," ");
                        while (st.hasMoreTokens()){
                              String ruleToken = st.nextToken();
                              NonTerminal ruleTokenNonTerminal = new
NonTerminal(ruleToken);
                              if (!allSymbols.contains(ruleTokenNonTerminal))
                                    allSymbols.add(ruleTokenNonTerminal);
                              nonTerminal.addVariable(ruleToken);
                        }
                        System.out.println("==>"+nonTerminal);
          */
                        StringBuffer out = new StringBuffer();
                        out.append("package "+astPackageName+";\n\n");
                        out.append("public class ");
                        out.append(sequenceHelper.productionRuleName);
```

```
                        System.out.println("AM:
"+sequenceHelper.productionRuleName+":"+alternativeMap);
                        String superClass = (String)
alternativeMap.get(sequenceHelper.productionRuleName);
                        if (superClass!=null)
                              out.append(" extends "+superClass);
                        out.append("{\n");
                        for (int i=0;i<symbolList.size();i++){
                              Symbol symbol = (Symbol) symbolList.get(i);
                              out.append("\tpublic "+symbol.getVariableType()+"
"+symbol.getVariableName()+";\n");
                        }
                        out.append("\tpublic int line;\n\n");
                        out.append("\tpublic
"+sequenceHelper.productionRuleName+"(");
                        for (int i=0;i<symbolList.size();i++){
                              Symbol symbol = (Symbol) symbolList.get(i);
                              out.append(symbol.getVariableType()+"
"+symbol.getVariableName());
                              out.append(", ");
                        }
                        out.append("int line){\n");
                        for (int i=0;i<symbolList.size();i++){
                              Symbol symbol = (Symbol) symbolList.get(i);
                              out.append("\t\tthis."+symbol.getVariableName()+"
= "+symbol.getVariableName()+";\n");
                        }
                        out.append("\t\tthis.line = line;\n");
                        out.append("\t}\n\n");
                        out.append("\tpublic String toString(){\n\t\treturn
\""+sequenceHelper.productionRuleName+"[ ");
                        for (int i=0;i<symbolList.size();i++){
                              Symbol symbol = (Symbol) symbolList.get(i);
                              out.append("\"+"+symbol.getVariableName()+"+\"");
                              if (i!=symbolList.size()-1)
                                    out.append(",");
                        }
                        out.append("]\";\n");
                        out.append("\t}\n\n");

                        out.append("\t\tpublic Object visit(Visitor v, Object
arg){\n\treturn v.visit"+sequenceHelper.productionRuleName+"(this, arg,
line);\n\t}\n\n");
                        out.append("}");
                  return new
GeneratedFile(sequenceHelper.productionRuleName+".java",astPackageName,out.toStr
ing());
      }

      private GeneratedFile generateVisitor(){
            StringBuffer out = new StringBuffer();
            out.append("package "+astPackageName+";\n\npublic interface
Visitor{\n\n");
            for (int i=0;i<allSymbols.size();i++){
                  String symbolName = (String) allSymbols.get(i);
```

```
                  out.append("\tpublic Object
visit"+symbolName+"("+symbolName+" "+symbolName.toLowerCase()+", Object arg, int
line);\n");
              }

            out.append("\n}");
            return new
GeneratedFile("Visitor.java",astPackageName,out.toString());

      }

      private List generateUndefinedNonTerminals(){
            List astList = new ArrayList();

            for (int i=0;i<allSymbols.size();i++){
                  String symbol = (String) allSymbols.get(i);
                  boolean existsDefinition = false;
                  Iterator definedNonTerminalsIterator =
definedNonTerminals.iterator();
                  while (definedNonTerminalsIterator.hasNext()){
                        String definedSymbol = (String)
definedNonTerminalsIterator.next();
                        if (symbol.equals(definedSymbol))
                              existsDefinition=true;
                  }

                  if (!existsDefinition)
                        nonDefinedTerminals.add(symbol);
            }

            System.out.println("*****************allSymbols:"+allSymbols);

      System.out.println("*****************definedNonTerminals:"+definedNonTerm
inals);

      System.out.println("*****************nonDefinedTerminals:"+nonDefinedTerm
inals);
            Iterator iterator = nonDefinedTerminals.iterator();
            while (iterator.hasNext()){
                  String nonTerminal = (String) iterator.next();
                  StringBuffer out = new StringBuffer(); out.append("package
"+astPackageName+";\n\n");
                  out.append("public class ");
                  out.append(nonTerminal);

                  String superClass = (String) alternativeMap.get(nonTerminal);
                  if (superClass!=null)
                        out.append(" extends "+superClass);
                  out.append("{\n");
                  out.append("\tpublic String spelling;\n");
out.append("\tpublic int line;\n\n");
                  out.append("\tpublic "+nonTerminal+"(String spelling, int
line){\n");
                  out.append("\t\tthis.spelling = spelling;\n");
                  out.append("\t\tthis.line = line;\n");
                  out.append("\t}\n\n");
```

```
                  out.append("\tpublic String toString(){\n\t\treturn
\""+nonTerminal+"[ \"+spelling+\" ]\";\n");
                  out.append("\t}\n\n");
                  out.append("\t\tpublic Object visit(Visitor v, Object
arg){\n\treturn v.visit"+nonTerminal+"(this, arg, line);\n\t}\n\n");
                  out.append("}");
                  astList.add(new
GeneratedFile(nonTerminal+".java",astPackageName,out.toString()));
            }

              return astList;
      }

      private GeneratedFile generateDisplay(){
            StringBuffer out = new StringBuffer();
            out.append("package "+astPackageName+";\n\npublic class
Display{\n\n");
            out.append("\tfinal private int BLANKS = 2;");
            out.append("\n\tpublic Display(Object p){\n\t\tdisplay(p,
0);\n\t}");
            out.append("\n\tprivate static void spaces(int count){\n\t\t
for(int i = 1; i <= count; i++)\n\t\t\t   System.out.print(\" \");\n\t}");


            out.append("\n\tprivate void display(Object p, int count){\n\t\tif(p
== null) return;\n\t\t");
            out.append("String s =
p.getClass().getName();\n\t\tspaces(count);");
            out.append("\n\t\tif
(s.equals(\"java.lang.String\")){\n\t\t\tSystem.out.println(\"<\"+p+\">\");\n\t\
t} else {\n\t\t\tSystem.out.println(s);\n\t\t}");
            //non terminals
            Iterator definedNonTerminalsHelpersIterator =
definedNonTerminalsHelpers.iterator();
            while(definedNonTerminalsHelpersIterator.hasNext()){
                  Object helper = definedNonTerminalsHelpersIterator.next();
                  if (helper instanceof ASTSequenceHelper){
                        ASTSequenceHelper sequenceHelper = (ASTSequenceHelper)
helper;

      out.append("\n\t\tif(s.equals(\""+astPackageName+"."+sequenceHelper.produc
tionRuleName+"\")){");
                  for (int j=0;j<sequenceHelper.sequenceList.size();j++){
                        Symbol symbol = (Symbol)
sequenceHelper.sequenceList.get(j);
                        if (symbol.canBeEmpty())
                        out.append("\n\t\t\tif
((("+astPackageName+"."+sequenceHelper.productionRuleName+")p)."+symbol.getVaria
bleName()+"!=null)");

      out.append("\n\t\t\t\tdisplay((("+astPackageName+"."+sequenceHelper.produc
tionRuleName+")p)."+symbol.getVariableName()+", count+BLANKS);");
                  }

                  out.append("\n\t\t}");
                  }
```

```
             }

            //terminals
            Iterator nonDefinedTerminalsIterator =
nonDefinedTerminals.iterator();
            while(nonDefinedTerminalsIterator.hasNext()){
                  String nonTerminal = (String)
nonDefinedTerminalsIterator.next();

      out.append("\n\t\tif(s.equals(\""+astPackageName+"."+nonTerminal+"\")){");
                  out.append("\n\t\t\tspaces(count);");

      out.append("\n\t\t\tSystem.out.println(\"***\"+(("+nonTerminal+")p).spelli
ng+\"***\");");
                  out.append("\n\t\t}");
            }



            out.append("\n\t}");

            out.append("\n}");
            return new
GeneratedFile("Display.java",astPackageName,out.toString());

      }

      public GeneratedFile generateError(){
            StringBuffer out = new StringBuffer();
            out.append("package "+parserPackageName+";\n\npublic class
Error{\n\n");
            out.append("\tpublic Error(String message, int line){\n");
            out.append("\t\tSystem.out.println(\"Line \" + line + \": \" +
message);\n");
            out.append("\t\tSystem.exit(0);\n");
            out.append("\t}\n");
            out.append("\n}");

            return new
GeneratedFile("Error.java",parserPackageName,out.toString());
      }

      public GeneratedFile generateCompileBat(){
            StringBuffer out = new StringBuffer();
            out.append("javac -classpath .;log4j-1.2rc1.jar;common.jar
"+parserPackageName+"/Main.java");
            return new GeneratedFile("compile.bat",null,out.toString());
      }

      public GeneratedFile generateRunBat(){
            StringBuffer out = new StringBuffer();
            out.append("java -classpath .;log4j-1.2rc1.jar;common.jar
"+parserPackageName+".Main");
            return new GeneratedFile("run.bat",null,out.toString());
      }

      public GeneratedFile generateMain(){
```

```
            StringBuffer out = new StringBuffer();
            out.append("package "+parserPackageName+";\n\n");
//          out.append("import org.apache.log4j.*;\n");
            out.append("import java.util.*;\n\n");
            out.append("public class Main{\n\n");


            out.append("\tpublic static void main(String[] args){\n");
//          out.append("\t\tProperties props = new Properties();\n");
//          out.append("\t\t//Set root logger level to DEBUG and its only
appender to A1.\n");
//          out.append("\t\tprops.put(\"log4j.rootLogger\",\"DEBUG,A1\");\n");
//          out.append("\t\t//A1 is set to be a ConsoleAppender.\n");
//
      out.append("\t\tprops.put(\"log4j.appender.A1\",\"org.apache.log4j.Console
Appender\");\n");
//          out.append("\t\t// A1 uses PatternLayout.\n");
//
      out.append("\t\tprops.put(\"log4j.appender.A1.layout\",\"org.apache.log4j.
PatternLayout\");\n");
//
      out.append("\t\tprops.put(\"log4j.appender.A1.layout.ConversionPattern\",\
"=*=>%-4r [%t] %-5p %c %x - %m%n\");\n");
//          out.append("\t\t//PropertyConfigurator.configure(props);\n");
            out.append("\t\tParser parser = new Parser();\n");
            out.append("\t\t"+astPackageName+".Program program =
parser.parse();\n");
            out.append("\t\t"+astPackageName+".Display display = new
"+astPackageName+".Display(program);\n");
            out.append("\t}\n");
            out.append("\n}");

            return new
GeneratedFile("Main.java",parserPackageName,out.toString());
      }


      public GeneratedFile generateSourceFile(){
            StringBuffer out = new StringBuffer();
            out.append("package "+parserPackageName+";\n");
            out.append("import java.io.*;\n");
            out.append("\npublic class SourceFile{\n\n");
            out.append("\tpublic BufferedReader openFile() {\n");
out.append("\t\tBufferedReader inFile = null;\n");
            out.append("\t\tinFile = new BufferedReader(new
InputStreamReader(ClassLoader.getSystemResourceAsStream(\""+parserPackageName+"/
test.txt\")));\n");
            out.append("\t\treturn inFile;\n");
            out.append("\t}\n");
            out.append("\n}");

            return new
GeneratedFile("SourceFile.java",parserPackageName,out.toString());
      }


      public GeneratedFile generateParser(){
```

```
            StringBuffer header = new StringBuffer();
            header.append("package "+parserPackageName+";\n\nimport
"+astPackageName+".*;\n"+
//                  "import org.apache.log4j.*;\n"+
                    "\n"+
                    "public class Parser{\n"+
//                  "\tpublic static Logger logger =
Logger.getLogger(\"Parser\");\n\n"+
                    "\tprivate Token currentToken;\n"+
                    "\tScanner scanner;\n\n"+
                    "\tprivate void accept(byte expectedKind){\n"+
              "\t\tif(currentToken.getKind() == expectedKind)\n"+
                    "\t\t\tcurrentToken = (Token) scanner.scan();\n"+
              "\t\telse\n"+
                    "\t\tnew Error(\"Syntax error: \" + currentToken.spelling + \"
is not expected.  Expected \"+expectedKind+\" but got
\"+currentToken.getKind()+\".\",currentToken.line);\n"+
                    "\t}\n\n"+
                    "\tprivate void acceptIt(){\n"+
              "\t\tcurrentToken = (Token) scanner.scan();\n"+
                    "\t}\n\n"+
                    "\tpublic Program parse(){\n"+
//                  "\tlogger.debug(\"parse()\");\n"+
              "\t\tSourceFile sourceFile = new SourceFile();\n"+
              "\t\tscanner = new Scanner(sourceFile.openFile());\n"+
              "\t\tcurrentToken = (Token) scanner.scan();\n"+
              "\t\tProgram program = parseProgram();\n"+
              "\t\tif(currentToken.getKind() != Token.EOT)\n"+
                    "\t\t\tnew Error(\"Syntax error: Redundant characters at the
end of program.\",currentToken.line);\n"+
                    "\t\treturn program;\n"+
                    "\t\t}\n\n");

                    Iterator undefinedNonTerminalSetIterator =
undefinedNonTerminalSet.iterator();
                  while (undefinedNonTerminalSetIterator.hasNext()){
                        String undefinedNonTerminal = (String)
undefinedNonTerminalSetIterator.next();
                        header.append(
                              "\tpublic "+undefinedNonTerminal+"
parse"+undefinedNonTerminal+"(){\n"+
// "\t\tlogger.debug(\"parse"+undefinedNonTerminal+"()\");\n"+
                              "\t\t"+undefinedNonTerminal+"
"+undefinedNonTerminal.toLowerCase()+"= new
"+undefinedNonTerminal+"(currentToken.spelling,currentToken.line);\n"+

      "\t\taccept(Token."+token.lookupConstantSpelling(undefinedNonTerminal)+");
\n"+
                              "\t\treturn
"+undefinedNonTerminal.toLowerCase()+";\n"+
                              "\t}\n\n"
                        );
                  }

            out.insert(0,header);
            out.append("\n}");
```

```
            return new
GeneratedFile("Parser.java",parserPackageName,out.toString());

      }


      private static void writeOutput(GeneratedFile astFile){

      writeOutput(astFile.getFilename(),astFile.getPackageName(),astFile.getCont
ent());
      }

      private static void writeOutput(String filename,String packageName,String
data){
            try{
                  String packageNamePath = packageName;
                  while (packageNamePath.indexOf(".")!=-1){
                        int location = packageNamePath.indexOf(".");
                        packageNamePath =
packageNamePath.substring(0,location)+"\\"+packageNamePath.substring(location+1,
packageNamePath.length());
                  }

                  FileWriter out = new FileWriter("C:\\Documents and
Settings\\jfoure\\Desktop\\cs605\\Thesis\\ebnf\\"+packageNamePath+"\\"+filename+
".java");
                  out.write(data);
                  out.flush();
                  out.close();
            } catch (Exception e) {
                  e.printStackTrace();
            }

      }

      public List getAstFileList(){
            return astFileList;
      }


      class DefinitionSequenceHelper{
            int count = 0;
            boolean isAlternative = false;
            boolean startsWithEmpty = true;

            /**
             * @return
             */
            public int getCount() {
                  return count;
            }

            /**
             * @return
             */
            public boolean isAlternative() {
                  return isAlternative;
```

```
		}

		/**
		 * @param i
		 */
		public void setCount(int i) {
			count = i;
		}

		/**
		 * @param b
		 */
		public void setAlternative(boolean b) {
			isAlternative = b;
		}

		public void incCount(){
			++count;
		}
		/**
		 * @return
		 */
		public boolean isStartsWithEmpty() {
			return startsWithEmpty;
		}

		/**
		 * @param b
		 */
		public void setStartsWithEmpty(boolean b) {
			startsWithEmpty = b;
		}

	}
}
```

ast2parser.ASTAlternativeHelper.java

```
/*
 * Created on Feb 21, 2004
 */
package ast2parser;
import java.util.*;
import astast.*;
/**
 * @author jfoure
 */
public class ASTAlternativeHelper {
	List alternativeList = new ArrayList();
	String productionRuleName = null;
	boolean isBasic = false;

	public ASTAlternativeHelper(String name){
		this(name,false);
```

```
	}
	public ASTAlternativeHelper(String name,boolean isBasic){
		this.productionRuleName = name;
		this.isBasic = isBasic;
	}

	public void addAlternative(Symbol alternative){
		alternativeList.add(alternative);
	}

	public String toString(){
		return "ASTAlternativeHelper
["+productionRuleName+","+isBasic+","+alternativeList+"]";
	}
	/**
	 * @return
	 */
	public boolean isBasic() {
		return isBasic;
	}

	/**
	 * @param b
	 */
	public void setBasic(boolean b) {
		isBasic = b;
	}

}
```

ast2parser.AstGenerator.java

```
/*
 * AstGenerator.java
 *
 * Created on January 31, 2004, 2:32 PM
 */

package ast2parser;
import java.io.*;
import java.util.*;

import astast.*;
import common.Error;
import common.SourceFile;
/**
 *
 * @author  jfoure
 * @version
 */
public class AstGenerator {
	private String packageName;
    private Map alternativeMap = new Hashtable(); private List allSymbols = new
ArrayList();
	private Collection definedNonTerminals = new ArrayList();
```

```
      private Collection nonDefinedTerminals = new ArrayList();
    /** Creates new AstGenerator */
    public AstGenerator(String packageName) {
      this.packageName = packageName;
    }

    /*
    generateAst();
    try{
//          BufferedReader inFile = new BufferedReader(new
FileReader("C:\\Documents and
Settings\\jfoure\\Desktop\\cs605\\Thesis\\ebnfast.txt"));
            BufferedReader inFile = new BufferedReader(new
FileReader("C:\\Documents and
Settings\\jfoure\\Desktop\\cs605\\Thesis\\astast.txt"));
    String line = null;
        while ( (line=inFile.readLine()) !=null){
            System.out.println(line);
            if (line.indexOf("::=")!=-1){
                        NonTerminal nonTerminal = new
NonTerminal(line.substring(0,line.indexOf("::=")).trim());
                        definedNonTerminals.add(nonTerminal);
                String rule =
line.substring(line.indexOf("::=")+3,line.length()-1).trim();
                if (rule.indexOf('|')!=-1)
                    generateAlternatives(nonTerminal,rule);
                else{
                    generateDefinitions(nonTerminal,rule);
                }

            }

        }
    }
    catch(IOException e){
        System.out.println(e);
    }

      generateVisitor();
      System.out.println("allNonTerminals:"+allSymbols);
      System.out.println("definedNonTerminals:"+definedNonTerminals);
      System.out.println("**************************");
//    allSymbols.removeAll(definedNonTerminals);
//    System.out.println("intersect:"+allSymbols);
//    generateUndefinedNonTerminals(allSymbols);
      for (int i=0;i<allSymbols.size();i++){
            NonTerminal nonTerminal = (NonTerminal) allSymbols.get(i);

            boolean existsDefinition = false;
            Iterator definedNonTerminalsIterator =
definedNonTerminals.iterator();
            while (definedNonTerminalsIterator.hasNext()){
                  NonTerminal definedNonTerminal = (NonTerminal)
definedNonTerminalsIterator.next();
                  if
(nonTerminal.getName().equals(definedNonTerminal.getName()))
                        existsDefinition=true;
```

```
            }

            if (!existsDefinition)
                  nonDefinedTerminals.add(nonTerminal);
      }

      System.out.println("*************************");
      System.out.println("intersect:"+nonDefinedTerminals);
      generateUndefinedNonTerminals(nonDefinedTerminals);
      generateDisplay();
      generateChecker();

    }
*/


    /*









      private void generateChecker(){
            StringBuffer out = new StringBuffer();
            out.append("package astast;\n\nimport astscanner.*;\nimport
astparser.*;\n\npublic class Checker implements Visitor{\n\n");

            //non terminals
            int count = 0;
            Iterator definedNonTerminalsIterator =
definedNonTerminals.iterator();
            while(definedNonTerminalsIterator.hasNext()){
                  NonTerminal nonTerminal = (NonTerminal)
definedNonTerminalsIterator.next();
                  if (count==0){
                        out.append("\n\tpublic void
check("+nonTerminal.getName()+" "+nonTerminal.getName().toLowerCase()+"){");

      out.append("\n\t\t"+nonTerminal.getName().toLowerCase()+".visit(this,
null);");
                        out.append("\n\t}");
                  }
                  ++count;
                  out.append("\n\tpublic Object
visit"+nonTerminal.getName()+"("+nonTerminal.getName()+"
"+nonTerminal.getName().toLowerCase()+", Object arg, int line){\n");
                  for (int j=0;j<nonTerminal.getVariableList().size();j++){
                        Variable variable =
(Variable)nonTerminal.getVariableList().get(j);
                        if (!variable.getName().endsWith("Tail")){

      out.append("\n\t\t"+nonTerminal.getName().toLowerCase()+"."+variable.getNa
me()+".visit(this,null);");
                        } else {
```

```
                              out.append("\n\t\tif
("+nonTerminal.getName().toLowerCase()+"."+variable.getName()+"!=null)");

      out.append("\n\t\t\t"+nonTerminal.getName().toLowerCase()+"."+variable.get
Name()+".visit(this,null);");
                        }



      //out.append("\n\t\t\tdisplay((("+nonTerminal.getName()+")p)."+variable.ge
tName()+", count+BLANKS);");
                  }
                  out.append("\n\t\treturn null;");
                  out.append("\n\t}");
             }

            //terminals
            Iterator nonDefinedTerminalsIterator =
nonDefinedTerminals.iterator();
            while(nonDefinedTerminalsIterator.hasNext()){
                  NonTerminal nonTerminal = (NonTerminal)
nonDefinedTerminalsIterator.next();
                  out.append("\n\tpublic Object
visit"+nonTerminal.getName()+"("+nonTerminal.getName()+"
"+nonTerminal.getName().toLowerCase()+", Object arg, int line){");

      //out.append("\n\t\t"+nonTerminal.getName().toLowerCase()+"."+variable.get
Name()+".visit(this,null);");
                  out.append("\n\t\treturn null;");
                  out.append("\n\t}");
            }



            out.append("\n");

            out.append("\n\tpublic static void main(String[] args){");
            out.append("\n\t\tParser ebnfparser = new Parser();");
            out.append("\n\t\tGrammar grammar = ebnfparser.parse();");
            out.append("\n\t\tnew Display(grammar);");
            out.append("\n\t\tChecker check = new Checker();");
            out.append("\n\t\tcheck.check(grammar);");
            out.append("\n\t\tSystem.out.println(\"The program is terminated
normally.\");");
            out.append("\n\t\t}");

            out.append("\n}");

            writeOutput("Checker",out.toString());

      }


*/
```

```
/*



*/
    private boolean isLiteral(String token){
        return token.charAt(0)=='“' || token.charAt(0)=='”' ||
token.charAt(0)=='"';
    }

    public static void main(String args[]){
        AstGenerator astGenerator = new AstGenerator("emtgenerated.ast");
    }

    private void writeOutput(String filename, String data){
        try{
            String packageNamePath = packageName;
            while (packageNamePath.indexOf(".")!=-1){
                  int location = packageNamePath.indexOf(".");
                        packageNamePath =
packageNamePath.substring(0,location)+"\\"+packageNamePath.substring(location+1,
packageNamePath.length());
            }
            System.out.println("packageNamePath:"+packageNamePath);
            FileWriter out = new FileWriter("C:\\Documents and
Settings\\jfoure\\Desktop\\cs605\\Thesis\\ebnf\\"+packageNamePath+"\\"+filename+
".java");
            out.write(data);
            out.flush();
            out.close();
        } catch (Exception e) {
            e.printStackTrace();
        }

    }



}
```

ast2parser.ASTSequenceHelper.java

```
/*
 * Created on Feb 21, 2004
 */
package ast2parser;
import java.util.*;
import astast.*;
/**
 * @author jfoure
 */
public class ASTSequenceHelper {
      List sequenceList = new ArrayList();
      String productionRuleName = null;
```

```
	public ASTSequenceHelper(String name){
		this.productionRuleName = name;
	}

	public void addSequence(Symbol symbol){
		sequenceList.add(symbol);
	}

	public String toString(){
		return "ASTSequenceHelper
["+productionRuleName+","+sequenceList+"]";
	}
}
```

ast2parser.NonTerminalChecker.java

```
package ast2parser;
import astast.*;
import ebnfscanner.*;
import ebnfparser.*;

import java.util.*;
public class NonTerminalChecker implements Visitor{
	Set definedNonTerminalSet = new HashSet();
	Set nonTerminalSet = new HashSet();
	public Set getDefinedNonTerminalSet(){
		return definedNonTerminalSet;
	}
	public Set getUndefinedNonTerminalSet(){
		return nonTerminalSet;
	}




	public void check(Grammar grammar){
		grammar.visit(this, null);
		nonTerminalSet.removeAll(definedNonTerminalSet);
	}
	public Object visitGrammar(Grammar grammar, Object arg, int line){
		grammar.productionRuleList.visit(this,null); return null;
	}
	public Object visitProductionRuleList(ProductionRuleList
productionrulelist, Object arg, int line){
		productionrulelist.productionRule.visit(this,null); if
(productionrulelist.productionRuleListTail!=null)
			productionrulelist.productionRuleListTail.visit(this,null);
		return null;
	}
	public Object visitProductionRuleListTail(ProductionRuleListTail
productionrulelisttail, Object arg, int line){
		productionrulelisttail.productionRule.visit(this,null); if
(productionrulelisttail.productionRuleListTail!=null)

	productionrulelisttail.productionRuleListTail.visit(this,null);
```

```
            return null;
      }
      public Object visitProductionRule(ProductionRule productionrule, Object
arg, int line){
            definedNonTerminalSet.add(productionrule.nonTerminal.spelling);
            productionrule.nonTerminal.visit(this,null);
productionrule.definition.visit(this,null); return null;
      }
      public Object visitDefinition(Definition definition, Object arg, int
line){
            definition.symbol.visit(this,null); if
(definition.alternativeDefinition!=null)
                  definition.alternativeDefinition.visit(this,null);
            if (definition.sequenceDefinition!=null)
                  definition.sequenceDefinition.visit(this,null);
            return null;
      }
      public Object visitAlternativeDefinition(AlternativeDefinition
alternativedefinition, Object arg, int line){
            alternativedefinition.symbol.visit(this,null); if
(alternativedefinition.alternativeDefinition!=null)
                  alternativedefinition.alternativeDefinition.visit(this,null);
            return null;
      }
      public Object visitSequenceDefinition(SequenceDefinition
sequencedefinition, Object arg, int line){
            sequencedefinition.symbol.visit(this,null); if
(sequencedefinition.sequenceDefinition!=null)
                  sequencedefinition.sequenceDefinition.visit(this,null);
            return null;
      }
      public Object visitNonTerminal(NonTerminal nonterminal, Object arg, int
line){
            nonTerminalSet.add(nonterminal.spelling);
            return null;
      }
      public Object visitTerminalSymbol(TerminalSymbol terminalsymbol, Object
arg, int line){
            terminalsymbol.terminal.visit(this,null);
            return null;
      }
      public Object visitTerminal(Terminal terminal, Object arg, int line){
            return null;
      }
      public Object visitNonTerminalSymbol(NonTerminalSymbol nonterminalsymbol,
Object arg, int line){
            nonterminalsymbol.nonTerminal.visit(this,null);
            return null;

      }


      public static void main(String[] args) throws Exception{
            Parser parser = new Parser();
            ebnfast.Grammar ebnfGrammar = parser.parse();
            ebnf2ast.EBNF2ASTConverter ebnf2ASTConverter = new
ebnf2ast.EBNF2ASTConverter(new emt.Token());
```

```
            ebnf2ASTConverter.check(ebnfGrammar);
            Grammar astGrammar = ebnf2ASTConverter.getAstGrammar();
//          new Display(grammar);
            System.out.println("*****");
            NonTerminalChecker nonTerminalChecker = new NonTerminalChecker();
            nonTerminalChecker.check(astGrammar);


      System.out.println("definedNonTerminalSet:"+nonTerminalChecker.getDefinedN
onTerminalSet());

      System.out.println("nonTerminalSet:"+nonTerminalChecker.getUndefinedNonTer
minalSet());

      }



}
```

astast.AlternativeDefinition.java

```
package astast;
public class AlternativeDefinition{
      public Symbol symbol;
      public AlternativeDefinition alternativeDefinition;
      public int line;
      public AlternativeDefinition(Symbol symbol, AlternativeDefinition
alternativeDefinition, int line){
            this.symbol = symbol;
            this.alternativeDefinition = alternativeDefinition;
            this.line = line;
      }

      public String toString(){
            return "AlternativeDefinition[
"+symbol+","+alternativeDefinition+"]";
      }

            public Object visit(Visitor v, Object arg){
      return v.visitAlternativeDefinition(this, arg, line);
      }

}
```

astast.Ast.java

```
package astast;
public abstract class Ast{
      public abstract Object visit(Visitor v, Object arg);
}
```

astast.Checker.java

```
package astast;
import astscanner.*;
import astparser.*;

public class Checker implements Visitor{

      public void check(Grammar grammar){
            grammar.visit(this, null);
      }
      public Object visitGrammar(Grammar grammar, Object arg, int line){
            grammar.productionRuleList.visit(this,null); return null;
      }
      public Object visitProductionRuleList(ProductionRuleList
productionrulelist, Object arg, int line){
            productionrulelist.productionRule.visit(this,null); if
(productionrulelist.productionRuleListTail!=null)
                  productionrulelist.productionRuleListTail.visit(this,null);
            return null;
      }
      public Object visitProductionRuleListTail(ProductionRuleListTail
productionrulelisttail, Object arg, int line){
            productionrulelisttail.productionRule.visit(this,null); if
(productionrulelisttail.productionRuleListTail!=null)

      productionrulelisttail.productionRuleListTail.visit(this,null);
            return null;
      }
      public Object visitProductionRule(ProductionRule productionrule, Object
arg, int line){
            productionrule.nonTerminal.visit(this,null);
productionrule.definition.visit(this,null); return null;
      }
      public Object visitDefinition(Definition definition, Object arg, int
line){
            definition.symbol.visit(this,null); if
(definition.alternativeDefinition!=null)
                  definition.alternativeDefinition.visit(this,null);
            if (definition.sequenceDefinition!=null)
                  definition.sequenceDefinition.visit(this,null);
            return null;
      }
      public Object visitAlternativeDefinition(AlternativeDefinition
alternativedefinition, Object arg, int line){
            alternativedefinition.symbol.visit(this,null); if
(alternativedefinition.alternativeDefinition!=null)
                  alternativedefinition.alternativeDefinition.visit(this,null);
            return null;
      }
      public Object visitSequenceDefinition(SequenceDefinition
sequencedefinition, Object arg, int line){
            sequencedefinition.symbol.visit(this,null); if
(sequencedefinition.sequenceDefinition!=null)
                  sequencedefinition.sequenceDefinition.visit(this,null);
            return null;
```

```
	}
	public Object visitTerminalSymbol(TerminalSymbol terminalsymbol, Object
arg, int line){
		terminalsymbol.terminal.visit(this,null);
		return null;
	}
	public Object visitTerminal(Terminal terminal, Object arg, int line){
		return null;
	}
	public Object visitNonTerminalSymbol(NonTerminalSymbol nonterminalsymbol,
Object arg, int line){
		nonterminalsymbol.nonTerminal.visit(this,null);
		return null;
	}
	public Object visitNonTerminal(NonTerminal nonterminal, Object arg, int
line){
		return null;
	}


	public static void main(String[] args){
		Parser ebnfparser = new Parser();
		Grammar grammar = ebnfparser.parse();
		new Display(grammar);
		Checker check = new Checker();
		check.check(grammar);
		System.out.println("The program is terminated normally.");
		}
}
```

astast.Definition.java

```
package astast;
public class Definition{
	public Symbol symbol;
	public AlternativeDefinition alternativeDefinition;
	public SequenceDefinition sequenceDefinition;
	public int line;
	public Definition(Symbol symbol, AlternativeDefinition
alternativeDefinition, SequenceDefinition sequenceDefinition, int line){
		this.symbol = symbol;
		this.alternativeDefinition = alternativeDefinition;
		this.sequenceDefinition = sequenceDefinition;
		this.line = line;
	}

	public String toString(){
		return "Definition[
"+symbol+","+alternativeDefinition+","+sequenceDefinition+"]";
	}

		public Object visit(Visitor v, Object arg){
	return v.visitDefinition(this, arg, line);
	}
```

```
}
```

astast.Display.java

```
package astast;
public class Display{
      final private int BLANKS = 2;
      public Display(Object p){
            display(p, 0);
      }
      private static void spaces(int count){
              for(int i = 1; i <= count; i++)
                        System.out.print(" ");
      }
      private void display(Object p, int count){
            if(p == null) return;
            String s = p.getClass().getName();
            spaces(count);
            System.out.println(s);
            if(s.equals("astast.Grammar")){
                  display(((Grammar)p).productionRuleList, count+BLANKS);
            }
            if(s.equals("astast.ProductionRuleList")){
                  display(((ProductionRuleList)p).productionRule, count+BLANKS);
                  display(((ProductionRuleList)p).productionRuleListTail,
count+BLANKS);
            }
            if(s.equals("astast.ProductionRuleListTail")){
                  display(((ProductionRuleListTail)p).productionRule,
count+BLANKS);
                  display(((ProductionRuleListTail)p).productionRuleListTail,
count+BLANKS);
            }
            if(s.equals("astast.ProductionRule")){
                  display(((ProductionRule)p).nonTerminal, count+BLANKS);
                  display(((ProductionRule)p).definition, count+BLANKS);
            }
            if(s.equals("astast.Definition")){
                  display(((Definition)p).symbol, count+BLANKS);
                  display(((Definition)p).alternativeDefinition, count+BLANKS);
                  display(((Definition)p).sequenceDefinition, count+BLANKS);
            }
            if(s.equals("astast.AlternativeDefinition")){
                  display(((AlternativeDefinition)p).symbol, count+BLANKS);
                  display(((AlternativeDefinition)p).alternativeDefinition,
count+BLANKS);
            }
            if(s.equals("astast.SequenceDefinition")){
                  display(((SequenceDefinition)p).symbol, count+BLANKS);
                  display(((SequenceDefinition)p).sequenceDefinition,
count+BLANKS);
            }
            if(s.equals("astast.NonTerminalSymbol")){
                  display(((NonTerminalSymbol)p).nonTerminal, count+BLANKS);
            }
```

```
            if(s.equals("astast.TerminalSymbol")){
                  display(((TerminalSymbol)p).terminal, count+BLANKS);
            }
            if(s.equals("astast.NonTerminal")){
                  spaces(count);
                  System.out.println("***"+((NonTerminal)p).spelling+"***");
            }
            if(s.equals("astast.Terminal")){
                  spaces(count);
                  System.out.println("\""+((Terminal)p).spelling+"\"");
            }
      }
}
```

astast.Grammar.java

```
package astast;
public class Grammar{
      public ProductionRuleList productionRuleList;
      public int line;
      public Grammar(ProductionRuleList productionRuleList, int line){
            this.productionRuleList = productionRuleList;
            this.line = line;
      }

      public String toString(){
            return "Grammar[ "+productionRuleList+"]";
      }

            public Object visit(Visitor v, Object arg){
      return v.visitGrammar(this, arg, line);
      }

}
```

astast.HtmlDisplay.java

```
package astast;
import java.io.*;
public class HtmlDisplay{
      final private static int BLANKS = 2;
      public HtmlDisplay(Object p){
      }
      private static void spaces(int count, Writer out) throws IOException{
              for(int i = 1; i <= count; i++)
                        out.write(" ");
      }

      public static void display(Grammar grammar, Writer out) throws
IOException{
            display(grammar, out, 0);
      }
```

```
      private static void display(Object p, Writer out, int count) throws
IOException{
            if(p == null) return;
            String s = p.getClass().getName();
            spaces(count, out);
            out.write(s+"<br/>\n");
            if(s.equals("astast.Grammar")){
                  display(((Grammar)p).productionRuleList, out, count+BLANKS);
            }
            if(s.equals("astast.ProductionRuleList")){
                  display(((ProductionRuleList)p).productionRule, out,
count+BLANKS);
                  display(((ProductionRuleList)p).productionRuleListTail, out,
count+BLANKS);
            }
            if(s.equals("astast.ProductionRuleListTail")){
                  display(((ProductionRuleListTail)p).productionRule, out,
count+BLANKS);
                  display(((ProductionRuleListTail)p).productionRuleListTail,
out, count+BLANKS);
            }
            if(s.equals("astast.ProductionRule")){
                  display(((ProductionRule)p).nonTerminal, out, count+BLANKS);
                  display(((ProductionRule)p).definition, out, count+BLANKS);
            }
            if(s.equals("astast.Definition")){
                  display(((Definition)p).symbol, out, count+BLANKS);
                  display(((Definition)p).alternativeDefinition, out,
count+BLANKS);
                  display(((Definition)p).sequenceDefinition, out,
count+BLANKS);
            }
            if(s.equals("astast.AlternativeDefinition")){
                  display(((AlternativeDefinition)p).symbol, out, count+BLANKS);
                  display(((AlternativeDefinition)p).alternativeDefinition, out,
count+BLANKS);
            }
            if(s.equals("astast.SequenceDefinition")){
                  display(((SequenceDefinition)p).symbol, out, count+BLANKS);
                  display(((SequenceDefinition)p).sequenceDefinition, out,
count+BLANKS);
            }
            if(s.equals("astast.NonTerminalSymbol")){
                  display(((NonTerminalSymbol)p).nonTerminal, out,
count+BLANKS);
            }
            if(s.equals("astast.TerminalSymbol")){
                  display(((TerminalSymbol)p).terminal, out, count+BLANKS);
            }
            if(s.equals("astast.NonTerminal")){
                  spaces(count,out);
                  out.write("***"+((NonTerminal)p).spelling+"***");
            }
            if(s.equals("astast.Terminal")){
                  spaces(count,out);
                  out.write("\""+((Terminal)p).spelling+"\"");
            }
```

```
	}
}
```

astast.HtmlLineDisplay.java

```
package astast;
import java.io.*;
public class HtmlLineDisplay{
	final private static int BLANKS = 2;
	public HtmlLineDisplay(Object p){
	}
	private static void spaces(int count, Writer out) throws IOException{
		for(int i = 1; i <= count; i++)
			out.write(" ");
	}

	public static void display(Grammar grammar, Writer out) throws
IOException{
		display(grammar, out, 0);
	}

	private static void display(Object p, Writer out, int count) throws
IOException{
		if(p == null) return;
		String s = p.getClass().getName();
		//spaces(count);
		//System.out.println(s);
		if(s.equals("astast.Grammar")){
			display(((Grammar)p).productionRuleList, out, count+BLANKS);
		}
		if(s.equals("astast.ProductionRuleList")){
			display(((ProductionRuleList)p).productionRule, out,
count+BLANKS);
			display(((ProductionRuleList)p).productionRuleListTail, out,
count+BLANKS);
		}
		if(s.equals("astast.ProductionRuleListTail")){
			display(((ProductionRuleListTail)p).productionRule, out,
count+BLANKS);
			display(((ProductionRuleListTail)p).productionRuleListTail,
out, count+BLANKS);
		}
		if(s.equals("astast.ProductionRule")){
			display(((ProductionRule)p).nonTerminal, out, count+BLANKS);
			out.write(" ::= ");
			display(((ProductionRule)p).definition, out, count+BLANKS);
			out.write(" .\n<br/>");
		}
		if(s.equals("astast.Definition")){
			display(((Definition)p).symbol, out, count+BLANKS);
			display(((Definition)p).alternativeDefinition, out,
count+BLANKS);
			display(((Definition)p).sequenceDefinition, out,
count+BLANKS);
```

```
            }
            if(s.equals("astast.AlternativeDefinition")){
                  out.write(" | ");
                  display(((AlternativeDefinition)p).symbol, out, count+BLANKS);
                  display(((AlternativeDefinition)p).alternativeDefinition, out,
count+BLANKS);
            }
            if(s.equals("astast.SequenceDefinition")){
                  out.write(" ");
                  display(((SequenceDefinition)p).symbol, out, count+BLANKS);
                  display(((SequenceDefinition)p).sequenceDefinition, out,
count+BLANKS);
            }
            if(s.equals("astast.NonTerminalSymbol")){
                  display(((NonTerminalSymbol)p).nonTerminal, out,
count+BLANKS);
            }
            if(s.equals("astast.TerminalSymbol")){
                  display(((TerminalSymbol)p).terminal, out, count+BLANKS);
            }
            if(s.equals("astast.NonTerminal")){
                  //spaces(count,out);
                  out.write(((NonTerminal)p).spelling);
            }
            if(s.equals("astast.Terminal")){
                  //spaces(count,out);
                  out.write("\""+((Terminal)p).spelling+"\"");
            }
      }
}
```

astast.LineDisplay.java

```
package astast;
public class LineDisplay{
      final private int BLANKS = 2;
      public LineDisplay(Object p){
            display(p, 0);
      }
      private static void spaces(int count){
              for(int i = 1; i <= count; i++)
                        System.out.print(" ");
      }
      private void display(Object p, int count){
            if(p == null) return;
            String s = p.getClass().getName();
            //spaces(count);
            //System.out.println(s);
            if(s.equals("astast.Grammar")){
                  display(((Grammar)p).productionRuleList, count+BLANKS);
            }
            if(s.equals("astast.ProductionRuleList")){
                  display(((ProductionRuleList)p).productionRule, count+BLANKS);
                  display(((ProductionRuleList)p).productionRuleListTail,
count+BLANKS);
```

```
            }
            if(s.equals("astast.ProductionRuleListTail")){
                  display(((ProductionRuleListTail)p).productionRule,
count+BLANKS);
                  display(((ProductionRuleListTail)p).productionRuleListTail,
count+BLANKS);
            }
            if(s.equals("astast.ProductionRule")){
//                System.out.print(((ProductionRule)p).nonTerminal.spelling+"
::= ");
                  display(((ProductionRule)p).nonTerminal, count+BLANKS);
                  System.out.print(" ::= ");
                  display(((ProductionRule)p).definition, count+BLANKS);
                  System.out.println(" .");
            }
            if(s.equals("astast.Definition")){
                  //System.out.print(((Definition)p).symbol.getSpelling());
                  display(((Definition)p).symbol, count+BLANKS);
                  display(((Definition)p).alternativeDefinition, count+BLANKS);
                  display(((Definition)p).sequenceDefinition, count+BLANKS);
            }
            if(s.equals("astast.AlternativeDefinition")){
                  //System.out.print(" |
"+((AlternativeDefinition)p).symbol.getSpelling());
                  System.out.print(" | ");
                  display(((AlternativeDefinition)p).symbol, count+BLANKS);
                  display(((AlternativeDefinition)p).alternativeDefinition,
count+BLANKS);
            }
            if(s.equals("astast.SequenceDefinition")){
                  //System.out.print("
"+((SequenceDefinition)p).symbol.getSpelling());
                  System.out.print(" ");
                  display(((SequenceDefinition)p).symbol, count+BLANKS);
                  display(((SequenceDefinition)p).sequenceDefinition,
count+BLANKS);
            }
            if(s.equals("astast.NonTerminalSymbol")){
                  display(((NonTerminalSymbol)p).nonTerminal, count+BLANKS);
            }
            if(s.equals("astast.TerminalSymbol")){
                  display(((TerminalSymbol)p).terminal, count+BLANKS);
            }
            if(s.equals("astast.NonTerminal")){
                  //spaces(count);
                  //System.out.println("***"+((NonTerminal)p).spelling+"***");
                  System.out.print(((NonTerminal)p).spelling);
            }
            if(s.equals("astast.Terminal")){
                  //spaces(count);
                  //System.out.println("\""+((Terminal)p).spelling+"\"");
                  System.out.print("\""+((Terminal)p).spelling+"\"");
            }

      }
}
```

astast.NonTerminal.java

```
package astast;
public class NonTerminal{
      public String spelling;
      public int line;

      public NonTerminal(String spelling, int line){
            this.spelling = spelling;
            this.line = line;
      }

      public String toString(){
            return "NonTerminal[ "+spelling+" ]";
      }

            public Object visit(Visitor v, Object arg){
      return v.visitNonTerminal(this, arg, line);
      }

}
```

astast.NonTerminalSymbol.java

```
package astast;
public class NonTerminalSymbol extends Symbol{
      public NonTerminal nonTerminal;
      public String variableType;
      public String variableName;
      public boolean canBeEmpty;
      public int line;

      public NonTerminalSymbol(NonTerminal nonTerminal, String variableType,
String variableName, boolean canBeEmpty, int line){
            this.nonTerminal = nonTerminal;
            this.variableType = variableType;
            this.variableName = variableName;
            this.canBeEmpty = canBeEmpty;
            this.line = line;
      }

      public String toString(){
            return "NonTerminalSymbol[ "+nonTerminal+"]";
      }

            public Object visit(Visitor v, Object arg){
      return v.visitNonTerminalSymbol(this, arg, line);
      }
      public String getSpelling(){
            return nonTerminal.spelling;
      }
      public boolean canBeEmpty(){
            return canBeEmpty;
```

```
	}

	/**
	 * @return
	 */
	public String getVariableName() {
		return variableName;
	}

	/**
	 * @param string
	 */
	public void setVariableName(String string) {
		variableName = string;
	}

	/**
	 * @return
	 */
	public String getVariableType() {
		return variableType;
	}

	/**
	 * @param string
	 */
	public void setVariableType(String string) {
		variableType = string;
	}

}
```

astast.ProductionRule.java

```
package astast;
public class ProductionRule{
	public NonTerminal nonTerminal;
	public Definition definition;
	public int line;

	public ProductionRule(NonTerminal nonTerminal, Definition definition, int
line){
		this.nonTerminal = nonTerminal;
		this.definition = definition;
		this.line = line;
	}

	public String toString(){
		return "ProductionRule[ "+nonTerminal+","+definition+"]";
	}

		public Object visit(Visitor v, Object arg){
	return v.visitProductionRule(this, arg, line);
	}
```

```
}
```

astast.ProductionRuleList.java

```
package astast;
public class ProductionRuleList{
      public ProductionRule productionRule;
      public ProductionRuleListTail productionRuleListTail;
      public int line;
      public ProductionRuleList(ProductionRule productionRule,
ProductionRuleListTail productionRuleListTail, int line){
            this.productionRule = productionRule;
            this.productionRuleListTail = productionRuleListTail;
            this.line = line;
      }

      public String toString(){
            return "ProductionRuleList[
"+productionRule+","+productionRuleListTail+"]";
      }

            public Object visit(Visitor v, Object arg){
      return v.visitProductionRuleList(this, arg, line);
      }

}
```

astast.ProductionRuleListTail.java

```
package astast;
public class ProductionRuleListTail{
      public ProductionRule productionRule;
      public ProductionRuleListTail productionRuleListTail;
      public int line;
      public ProductionRuleListTail(ProductionRule productionRule,
ProductionRuleListTail productionRuleListTail, int line){
            this.productionRule = productionRule;
            this.productionRuleListTail = productionRuleListTail;
            this.line = line;
      }

      public String toString(){
            return "ProductionRuleListTail[
"+productionRule+","+productionRuleListTail+"]";
      }

            public Object visit(Visitor v, Object arg){
      return v.visitProductionRuleListTail(this, arg, line);
      }

}
```

astast.SequenceDefinition.java

```
package astast;
public class SequenceDefinition{
      public Symbol symbol;
      public SequenceDefinition sequenceDefinition;
      public int line;
      public SequenceDefinition(Symbol symbol, SequenceDefinition
sequenceDefinition, int line){
            this.symbol = symbol;
            this.sequenceDefinition = sequenceDefinition;
            this.line = line;
      }

      public String toString(){
            return "SequenceDefinition[ "+symbol+","+sequenceDefinition+"]";
      }

            public Object visit(Visitor v, Object arg){
      return v.visitSequenceDefinition(this, arg, line);
      }

}
```

astast.Symbol.java

```
package astast;
public abstract class Symbol extends Ast{
      public abstract String getSpelling();
      public abstract boolean canBeEmpty();
      public abstract String getVariableName();
      public abstract String getVariableType();

}
```

astast.Terminal.java

```
package astast;
public class Terminal{
      public String spelling;
      public int line;

      public Terminal(String spelling, int line){
            System.out.println("===T===>"+spelling);
            this.spelling = spelling;
            this.line = line;
      }

      public String toString(){
            return "Terminal[ \""+spelling+"\" ]";
      }
```

```
            public Object visit(Visitor v, Object arg){
      return v.visitTerminal(this, arg, line);
      }

}
```

astast.TerminalSymbol.java

```
package astast;
public class TerminalSymbol extends Symbol{
      public Terminal terminal;
      public String variableType;
      public String variableName;
      public boolean canBeEmpty;
      public int line;

      public TerminalSymbol(Terminal terminal, String variableType, String
variableName, boolean canBeEmpty, int line){
            this.terminal = terminal;
            this.variableType = variableType;
            this.variableName = variableName;
            this.canBeEmpty = canBeEmpty;
            this.line = line;
      }

      public String toString(){
            return "TerminalSymbol[ "+terminal+"]";
      }

            public Object visit(Visitor v, Object arg){
      return v.visitTerminalSymbol(this, arg, line);
      }
      public String getSpelling(){
            return terminal.spelling;
      }
      public boolean canBeEmpty(){
            return canBeEmpty;
      }

      /**
       * @return
       */
      public String getVariableName() {
            return variableName;
      }

      /**
       * @param string
       */
      public void setVariableName(String string) {
            variableName = string;
      }

      /**
       * @return
```

```
	 */
	public String getVariableType() {
		return variableType;
	}

	/**
	 * @param string
	 */
	public void setVariableType(String string) {
		variableType = string;
	}

}
```

astast.Visitor.java

```
package astast;
public interface Visitor{
	public Object visitGrammar(Grammar grammar, Object arg, int line);
	public Object visitProductionRuleList(ProductionRuleList
productionrulelist, Object arg, int line);
	public Object visitProductionRule(ProductionRule productionrule, Object
arg, int line);
	public Object visitProductionRuleListTail(ProductionRuleListTail
productionrulelisttail, Object arg, int line);
	public Object visitNonTerminal(NonTerminal nonterminal, Object arg, int
line);
	public Object visitDefinition(Definition definition, Object arg, int
line);
	public Object visitAlternativeDefinition(AlternativeDefinition
alternativedefinition, Object arg, int line);
	public Object visitSequenceDefinition(SequenceDefinition
sequencedefinition, Object arg, int line);
	public Object visitTerminalSymbol(TerminalSymbol terminalsymbol, Object
arg, int line);
	public Object visitTerminal(Terminal terminal, Object arg, int line);
	public Object visitNonTerminalSymbol(NonTerminalSymbol nonterminalsymbol,
Object arg, int line);

}
```

astgenerator.AstGenerator.java

```
/*
 * AstGenerator.java
 *
 * Created on January 31, 2004, 2:32 PM
 */

package astgenerator;
import java.io.*;
import java.util.*;
```

```
import common.Error;
import common.SourceFile;
/**
 *
 * @author  jfoure
 * @version
 */
public class AstGenerator {
    private Map alternativeMap = new Hashtable();
    //private List nonTerminalList = new ArrayList();
    private List allSymbols = new ArrayList();
//    private Set definedNonTerminals = new HashSet();
      private Collection definedNonTerminals = new ArrayList();
      private Collection nonDefinedTerminals = new ArrayList();

    /** Creates new AstGenerator */
    public AstGenerator() {

    generateAst();

    try{
//          BufferedReader inFile = new BufferedReader(new
FileReader("C:\\Documents and
Settings\\jfoure\\Desktop\\cs605\\Thesis\\ebnfast.txt"));
            BufferedReader inFile = new BufferedReader(new
FileReader("C:\\Documents and
Settings\\jfoure\\Desktop\\cs605\\Thesis\\astast.txt"));
    String line = null;
        while ( (line=inFile.readLine()) !=null){
            System.out.println(line);
            if (line.indexOf("::=")!=-1){
                        NonTerminal nonTerminal = new
NonTerminal(line.substring(0,line.indexOf("::=")).trim());
                        definedNonTerminals.add(nonTerminal);
                String rule =
line.substring(line.indexOf("::=")+3,line.length()-1).trim();
                if (rule.indexOf('|')!=-1)
                    generateAlternatives(nonTerminal,rule);
                else{
                    generateDefinitions(nonTerminal,rule);
                }

            }

        }
    }
    catch(IOException e){
        System.out.println(e);
    }

      generateVisitor();
      System.out.println("allNonTerminals:"+allSymbols);
      System.out.println("definedNonTerminals:"+definedNonTerminals);
      System.out.println("*************************");
//    allSymbols.removeAll(definedNonTerminals);
//    System.out.println("intersect:"+allSymbols);
//    generateUndefinedNonTerminals(allSymbols);
```

```
      for (int i=0;i<allSymbols.size();i++){
            NonTerminal nonTerminal = (NonTerminal) allSymbols.get(i);

            boolean existsDefinition = false;
            Iterator definedNonTerminalsIterator =
definedNonTerminals.iterator();
            while (definedNonTerminalsIterator.hasNext()){
                  NonTerminal definedNonTerminal = (NonTerminal)
definedNonTerminalsIterator.next();
                  if
(nonTerminal.getName().equals(definedNonTerminal.getName()))
                        existsDefinition=true;
            }

            if (!existsDefinition)
                  nonDefinedTerminals.add(nonTerminal);
      }

      System.out.println("*************************");
      System.out.println("intersect:"+nonDefinedTerminals);
      generateUndefinedNonTerminals(nonDefinedTerminals);
      generateDisplay();
      generateChecker();

    }


    private void generateAst(){
        writeOutput("Ast","package astast;\n\npublic abstract class
Ast{\n\tpublic abstract Object visit(Visitor v, Object arg);\n}");
/*
public abstract class Ast{
  public abstract Object visit(Visitor v, Object arg);
}
 */
    }

    private void generateVisitor(){
        StringBuffer out = new StringBuffer();
        out.append("package astast;\n\npublic interface Visitor{\n\n");
        for (int i=0;i<allSymbols.size();i++){
                  Symbol symbol = (Symbol) allSymbols.get(i);
             out.append("\tpublic Object
visit"+symbol.getName()+"("+symbol.getName()+"
"+symbol.getName().toLowerCase()+", Object arg, int line);\n");
         }

        out.append("\n}");
        writeOutput("Visitor",out.toString());

    }


      private void generateDisplay(){
            StringBuffer out = new StringBuffer();
            out.append("package astast;\n\npublic class Display{\n\n");
        out.append("\tfinal private int BLANKS = 2;");
```

```
            out.append("\n\tpublic Display(Object p){\n\t\tdisplay(p,
0);\n\t}");
            out.append("\n\tprivate static void spaces(int count){\n\t\t
for(int i = 1; i <= count; i++)\n\t\t\t   System.out.print(\" \");\n\t}");

        out.append("\n\tprivate void display(Object p, int count){\n\t\tif(p ==
null) return;\n\t\tString s =
p.getClass().getName();\n\t\tspaces(count);\n\t\tSystem.out.println(s);");
            //non terminals
            Iterator definedNonTerminalsIterator =
definedNonTerminals.iterator();
            while(definedNonTerminalsIterator.hasNext()){
                  NonTerminal nonTerminal = (NonTerminal)
definedNonTerminalsIterator.next();

      out.append("\n\t\tif(s.equals(\"astast."+nonTerminal.getName()+"\")){");
                  for (int j=0;j<nonTerminal.getVariableList().size();j++){
                        Variable variable =
(Variable)nonTerminal.getVariableList().get(j);

      out.append("\n\t\t\tdisplay((("+nonTerminal.getName()+")p)."+variable.getN
ame()+", count+BLANKS);");

      //out.append("\n\t\t\tdisplay(((ActualParameterSequence)p).as,
count+BLANKS);");
                  }

                  out.append("\n\t\t}");
             }

            //terminals
            Iterator nonDefinedTerminalsIterator =
nonDefinedTerminals.iterator();
            while(nonDefinedTerminalsIterator.hasNext()){
                  NonTerminal nonTerminal = (NonTerminal)
nonDefinedTerminalsIterator.next();

      out.append("\n\t\tif(s.equals(\"astast."+nonTerminal.getName()+"\")){");
                  out.append("\n\t\t\tspaces(count);");

      out.append("\n\t\t\tSystem.out.println(\"***\"+(("+nonTerminal.getName()+"
)p).spelling+\"***\");");
                  out.append("\n\t\t}");
            }



            out.append("\n\t}");

            out.append("\n}");
            writeOutput("Display",out.toString());

      }
```

```
      private void generateChecker(){
            StringBuffer out = new StringBuffer();
            out.append("package astast;\n\nimport astscanner.*;\nimport
astparser.*;\n\npublic class Checker implements Visitor{\n\n");

            //non terminals
            int count = 0;
            Iterator definedNonTerminalsIterator =
definedNonTerminals.iterator();
            while(definedNonTerminalsIterator.hasNext()){
                  NonTerminal nonTerminal = (NonTerminal)
definedNonTerminalsIterator.next();
                  if (count==0){
                        out.append("\n\tpublic void
check("+nonTerminal.getName()+" "+nonTerminal.getName().toLowerCase()+"){");

      out.append("\n\t\t"+nonTerminal.getName().toLowerCase()+".visit(this,
null);");
                        out.append("\n\t}");
                  }
                  ++count;
                  out.append("\n\tpublic Object
visit"+nonTerminal.getName()+"("+nonTerminal.getName()+"
"+nonTerminal.getName().toLowerCase()+", Object arg, int line){\n");
                  for (int j=0;j<nonTerminal.getVariableList().size();j++){
                        Variable variable =
(Variable)nonTerminal.getVariableList().get(j);
                        if (!variable.getName().endsWith("Tail")){

      out.append("\n\t\t"+nonTerminal.getName().toLowerCase()+"."+variable.getNa
me()+".visit(this,null);");
                        } else {
                              out.append("\n\t\tif
("+nonTerminal.getName().toLowerCase()+"."+variable.getName()+"!=null)");

      out.append("\n\t\t\t"+nonTerminal.getName().toLowerCase()+"."+variable.get
Name()+".visit(this,null);");
                        }



      //out.append("\n\t\t\tdisplay((("+nonTerminal.getName()+")p)."+variable.ge
tName()+", count+BLANKS);");
                  }
                  out.append("\n\t\treturn null;");
                  out.append("\n\t}");
             }

            //terminals
            Iterator nonDefinedTerminalsIterator =
nonDefinedTerminals.iterator();
            while(nonDefinedTerminalsIterator.hasNext()){
                  NonTerminal nonTerminal = (NonTerminal)
nonDefinedTerminalsIterator.next();
                  out.append("\n\tpublic Object
visit"+nonTerminal.getName()+"("+nonTerminal.getName()+"
"+nonTerminal.getName().toLowerCase()+", Object arg, int line){");
```

```
      //out.append("\n\t\t"+nonTerminal.getName().toLowerCase()+"."+variable.get
Name()+".visit(this,null);");
                  out.append("\n\t\treturn null;");
                  out.append("\n\t}");
            }



            out.append("\n");

            out.append("\n\tpublic static void main(String[] args){");
            out.append("\n\t\tParser ebnfparser = new Parser();");
            out.append("\n\t\tGrammar grammar = ebnfparser.parse();");
            out.append("\n\t\tnew Display(grammar);");
            out.append("\n\t\tChecker check = new Checker();");
            out.append("\n\t\tcheck.check(grammar);");
            out.append("\n\t\tSystem.out.println(\"The program is terminated
normally.\");");
            out.append("\n\t\t}");

            out.append("\n}");

            writeOutput("Checker",out.toString());

      }



    private void generateAlternatives(NonTerminal nonTerminal, String rule){
                StringTokenizer st = new StringTokenizer(rule,"|");
                while (st.hasMoreTokens()){
                    String ruleToken = st.nextToken().trim();
                    alternativeMap.put(ruleToken,nonTerminal.getName());
                    System.out.println("==>"+ruleToken+" implements
"+nonTerminal.getName()+";");
                }
                writeOutput(nonTerminal.getName(),"package astast;\n\npublic
abstract class "+nonTerminal.getName()+" extends Ast{}");
    }

    private void generateDefinitions(NonTerminal nonTerminal, String rule){
            if (!allSymbols.contains(nonTerminal))
                  allSymbols.add(nonTerminal);

                StringTokenizer st = new StringTokenizer(rule," ");
                while (st.hasMoreTokens()){
                    String ruleToken = st.nextToken();
                    NonTerminal ruleTokenNonTerminal = new
NonTerminal(ruleToken);
                    if (!allSymbols.contains(ruleTokenNonTerminal))
                                    allSymbols.add(ruleTokenNonTerminal);
                    nonTerminal.addVariable(ruleToken);
                }
                System.out.println("==>"+nonTerminal);
```

```
                StringBuffer out = new StringBuffer(); out.append("package
astast;\n\n"); out.append("public class "); out.append(nonTerminal.getName());
                System.out.println("AM:
"+nonTerminal.getName()+":"+alternativeMap);
                String superClass = (String)
alternativeMap.get(nonTerminal.getName());
                if (superClass!=null)
                    out.append(" extends "+superClass);
                out.append("{\n");
                List variableList = nonTerminal.getVariableList();
                for (int i=0;i<variableList.size();i++){
                    Variable variable = (Variable) variableList.get(i);
                    out.append("\tpublic "+variable.getType()+"
"+variable.getName()+";\n");
                }
                out.append("\tpublic int line;\n\n");
                out.append("\tpublic "+nonTerminal.getName()+"(");
                        for (int i=0;i<variableList.size();i++){
                              Variable variable = (Variable)
variableList.get(i);
                    out.append(variable.getType()+" "+variable.getName());
                    out.append(", ");
                }
                out.append("int line){\n");
                        for (int i=0;i<variableList.size();i++){
                              Variable variable = (Variable)
variableList.get(i);
                    out.append("\t\tthis."+variable.getName()+" =
"+variable.getName()+";\n");
                }
                out.append("\t\tthis.line = line;\n");
                out.append("\t}\n\n");
                        out.append("\tpublic String toString(){\n\t\treturn
\""+nonTerminal.getName()+"[ ");
                        for (int i=0;i<variableList.size();i++){
                              Variable variable = (Variable)
variableList.get(i);
                              out.append("\"+"+variable.getName()+"+\"");
                              if (i!=variableList.size()-1)
                                    out.append(",");
                        }
                        out.append("]\";\n");
                        out.append("\t}\n\n");

                out.append("\t\tpublic Object visit(Visitor v, Object
arg){\n\treturn v.visit"+nonTerminal.getName()+"(this, arg, line);\n\t}\n\n");
                out.append("}");
writeOutput(nonTerminal.getName(),out.toString());
    }

    private void generateUndefinedNonTerminals(Collection allNonTerminals){
      Iterator iterator = allNonTerminals.iterator();
            while (iterator.hasNext()){
                  NonTerminal nonTerminal = (NonTerminal) iterator.next();
                  StringBuffer out = new StringBuffer(); out.append("package
astast;\n\n"); out.append("public class "); out.append(nonTerminal.getName());
```

```
                String superClass = (String)
alternativeMap.get(nonTerminal.getName());
                if (superClass!=null)
                      out.append(" extends "+superClass);
                out.append("{\n");
                out.append("\tpublic String spelling;\n");
out.append("\tpublic int line;\n\n");
                out.append("\tpublic "+nonTerminal.getName()+"(String
spelling, int line){\n");
                out.append("\t\tthis.spelling = spelling;\n");
                out.append("\t\tthis.line = line;\n");
                out.append("\t}\n\n");
                out.append("\tpublic String toString(){\n\t\treturn
\""+nonTerminal.getName()+"[ \"+spelling+\" ]\";\n");
                out.append("\t}\n\n");
                out.append("\t\tpublic Object visit(Visitor v, Object
arg){\n\treturn v.visit"+nonTerminal.getName()+"(this, arg, line);\n\t}\n\n");
                out.append("}");
writeOutput(nonTerminal.getName(),out.toString());
            }

    }

    private boolean isLiteral(String token){
        return token.charAt(0)=='"' || token.charAt(0)=='”' ||
token.charAt(0)=='"';
    }

    public static void main(String args[]){
        AstGenerator astGenerator = new AstGenerator();
    }

    private void writeOutput(String filename, String data){
        try{
            FileWriter out = new FileWriter("C:\\Documents and
Settings\\jfoure\\Desktop\\cs605\\Thesis\\ebnf\\astast\\"+filename+".java");
            out.write(data);
            out.flush();
            out.close();
        } catch (Exception e) {
            e.printStackTrace();
        }

    }



}
```

astgenerator.NonTerminal.java

```
/*
 * Created on Feb 4, 2004
 */
package astgenerator;
```

```
import java.util.*;
/**
 * @author jfoure
 */
public class NonTerminal implements Symbol{
	private String name;
	private List variableList = new ArrayList();
	public NonTerminal(String name){
		setName(name);
	}
	/**
	 * @return
	 */
	public String getName() {
		return name;
	}

	/**
	 * @param string
	 */
	public void setName(String string) {
		name = string;
	}

	/**
	 * @return
	 */
	public List getVariableList() {
		return variableList;
	}

	/**
	 * @param list
	 */
	public void setVariableList(List list) {
		variableList = list;
	}

	public void addVariable(String type){
		String name =null;
		if (type.length()==1)
			name = type.toLowerCase();
		else
			name = type.substring(0,1).toLowerCase()+type.substring(1);
		System.out.println("var: "+type+":"+name);
		int count = 1;
		for (int i=0;i<variableList.size();i++){
			Variable variable = (Variable)variableList.get(i);
			if (type.equals(variable.getType())){
				++count;
			}
		}

		if (count!=1){
			name = name+count;
		}
```

```
            Variable variable = new Variable(type,name);
            variableList.add(variable);
      }

      public String toString(){
            return "NonTerminal["+name+","+variableList+"]";
      }

      public static void main(String args[]){
            NonTerminal nonTerminal = new NonTerminal("Test");
            System.out.println(nonTerminal.variableList);
            nonTerminal.addVariable("NonTerminal");
            System.out.println(nonTerminal.variableList);
            nonTerminal.addVariable("A");
            System.out.println(nonTerminal.variableList);
            nonTerminal.addVariable("A");
            nonTerminal.addVariable("Terminal");
            System.out.println(nonTerminal.variableList);
            nonTerminal.addVariable("NonTerminal");
            System.out.println(nonTerminal.variableList);

      }
      /* (non-Javadoc)
       * @see java.lang.Object#equals(java.lang.Object)
       */
      public boolean equals(Object nonTerminalObject) {
            NonTerminal nonTerminal = (NonTerminal) nonTerminalObject;
            System.out.println("Equals:"+getName()+"/"+nonTerminal.getName());
            if (getName().equals(nonTerminal.getName()))
                  return true;
            else
                  return false;

      }

}
```

astgenerator.Symbol.java

```
/*
 * Created on Feb 7, 2004
 */
package astgenerator;

/**
 * @author jfoure
 */
public interface Symbol {
      String getName();
}
```

astgenerator.Terminal.java

```
/*
 * Created on Feb 7, 2004
 */
package astgenerator;

/**
 * @author jfoure
 */
public class Terminal implements Symbol {
	private String name;

	/**
	 * @return
	 */
	public String getName() {
		return name;
	}

	/**
	 * @param string
	 */
	public void setName(String string) {
		name = string;
	}

}
```

astgenerator.Variable.java

```
/*
 * Created on Feb 4, 2004
 */
package astgenerator;

/**
 * @author jfoure
 */
public class Variable {
	private String name;
	private String type;

	/**
	 *
	 */
	public Variable(String type, String name) {
		super();
		setType(type);
		setName(name);
	}

	/**
	 * @return
	 */
	public String getName() {
		return name;
```

```
	}

	/**
	 * @return
	 */
	public String getType() {
		return type;
	}

	/**
	 * @param string
	 */
	public void setName(String string) {
		name = string;
	}

	/**
	 * @param string
	 */
	public void setType(String string) {
		type = string;
	}

	public String toString(){
		return "Variable ["+type+","+name+"]";
	}
}
```

astparser.Parser.java

```
package astparser;
import astast.*;
import astscanner.*;
import common.Error;

public class Parser{
  private Token currentToken;
  Scanner scanner;

  private void accept(byte expectedKind){
    if(currentToken.kind == expectedKind){
		System.out.println("Accepted: "+currentToken);
      currentToken = scanner.scan();
      }
    else
      new Error("Syntax error: " + currentToken + " is not expected.  Expected
"+expectedKind,
        currentToken.line);
  }

  private void acceptIt(){
      System.out.println("Accepted: "+currentToken);
    currentToken = scanner.scan();
  }
```

```
  public Grammar parse(){
    SourceFile sourceFile = new SourceFile();
    scanner = new Scanner(sourceFile.openFile());
    currentToken = scanner.scan();
      Grammar grammar = parseGrammar();
    if(currentToken.kind != Token.EOT)
      new Error("Syntax error: Redundant characters at the end of program.",
        currentToken.line);
    return grammar;
  }

//Grammar         ::= ProductionRuleList .
private Grammar parseGrammar(){
      System.out.println(":"+currentToken.line+":"+currentToken.spelling+":"+cur
rentToken.kind+":Grammar");
      ProductionRuleList prl = parseProductionRuleList();
      return new Grammar(prl,currentToken.line);
}

//ProductionRuleList    ::= ProductionRule {ProductionRuleList} .
private ProductionRuleList parseProductionRuleList(){
      System.out.println(":"+currentToken.line+":"+currentToken.spelling+":"+cur
rentToken.kind+":ProductionRuleList");
      ProductionRule pr = parseProductionRule();
      ProductionRuleListTail prlt = null;
      ProductionRuleListTail current = null;
      ProductionRuleListTail prev = null;

      while (currentToken.kind != Token.EOT){
            ProductionRule pr2 = parseProductionRule();
            current = new ProductionRuleListTail(pr2,null,currentToken.line);
            if (prlt==null)
                  prlt = current;
            else
                  prev.productionRuleListTail = current;
            prev = current;

      }

      return new ProductionRuleList(pr,prlt,currentToken.line);
}

//ProductionRule  ::= NonTerminal "::=" Definition ".".
private ProductionRule parseProductionRule(){
      System.out.println(":"+currentToken.line+":"+currentToken.spelling+":"+cur
rentToken.kind+":***ProductionRule***");
      NonTerminal nt = parseNonTerminal();
      accept(Token.RULE);
      Definition d = parseDefinition();
      accept(Token.ENDRULE);

      return new ProductionRule(nt,d,currentToken.line);
}


//Definition  ::=             NonTerminal ( { "|" NonTerminal } | { NonTerminal}
).
```

```
private Definition parseDefinition(){
      System.out.println(":"+currentToken.line+":"+currentToken.spelling+":"+cur
rentToken.kind+":Definition");
      System.out.println("Start Definition");
      Symbol s = parseSymbol();
      AlternativeDefinition ad = null;
      AlternativeDefinition adcurrent = null;
      AlternativeDefinition adprev = null;

      SequenceDefinition sd = null;
      SequenceDefinition sdcurrent = null;
      SequenceDefinition sdprev = null;

      boolean foundDefinitionTail = false;
      while (currentToken.kind==Token.ALTERNATIVE){
            foundDefinitionTail = true;
            acceptIt();
            Symbol s1 = parseSymbol();

            adcurrent = new AlternativeDefinition(s1,null,currentToken.line);
            if (ad==null)
                  ad = adcurrent;
            else
                  adprev.alternativeDefinition = adcurrent;
            adprev = adcurrent;
      }

      if (foundDefinitionTail){
            if (currentToken.kind==Token.NONTERMINAL){
                  new Error("Found NONTERMINAL",currentToken.line);
            }
      } else {
            while (currentToken.kind==Token.NONTERMINAL){
                  foundDefinitionTail = true;
                  Symbol s2 = parseSymbol();

                  sdcurrent = new SequenceDefinition(s2,null,currentToken.line);
                  if (sd==null)
                        sd = sdcurrent;
                  else
                        sdprev.sequenceDefinition = sdcurrent;
                  sdprev = sdcurrent;
            }

      }

      System.out.println("End Definition");
      return new Definition(s,ad,sd,currentToken.line);
}



//Symbol          ::= Terminal | NonTerminal .
private Symbol parseSymbol(){
      System.out.println(":"+currentToken.line+":"+currentToken.spelling+":"+cur
rentToken.kind+":Symbol");
      if (currentToken.kind==Token.NONTERMINAL){
```

```
            NonTerminal nt = parseNonTerminal();
            return new NonTerminalSymbol(nt,null,null,false,currentToken.line);
      }
      else if (currentToken.kind==Token.TERMINAL){
            Terminal t = parseTerminal();
            return new TerminalSymbol(t,null,null,false,currentToken.line);
      }
      else
            return null;//throw new ParserException("Symbol expected instead of
"+currentToken+".", currentToken.line);

}

//NonTerminal           ::= NONTERMINAL .
private NonTerminal parseNonTerminal(){
      System.out.println(":"+currentToken.line+":"+currentToken.spelling+":"+cur
rentToken.kind+":NonTerminal");
      if (currentToken.kind!=Token.NONTERMINAL){
            //throw new ParserException("Expected
NonTerminal.",currentToken.line);
            return null;
      }
      else
      {
            NonTerminal nt = new
NonTerminal(currentToken.spelling,currentToken.line);
            acceptIt();
            return nt;
      }

}

//Terminal        ::= " String " .
private Terminal parseTerminal(){
      System.out.println(":"+currentToken.line+":"+currentToken.spelling+":"+cur
rentToken.kind+":Terminal");
      if (currentToken.kind!=Token.TERMINAL){
            //throw new ParserException("Expected Terminal.",currentToken.line);
            return null;
      }
      else
      {
            Terminal t = new Terminal(currentToken.spelling,currentToken.line);
            acceptIt();
            return t;
      }

}

  public static void main(String[] args){
      System.out.println("Starting");
    Parser parser = new Parser();
    Grammar grammar = parser.parse();
      System.out.println("**************************");
      System.out.println(grammar);
      System.out.println("**************************");
      new Display(grammar);
```

```
      System.out.println("*************************");
      new LineDisplay(grammar);
      System.out.println("Stopping");
  }

}
```

astscanner.Scanner.java

```
package astscanner;
import java.io.*;
import common.Error;
public class Scanner{
  private char currentChar;
  private byte currentKind;
  private StringBuffer currentSpelling;
  private BufferedReader inFile;
  private static int line = 1;

  public Scanner(BufferedReader inFile){
    this.inFile = inFile;
    try{
      int i = this.inFile.read();
      if(i == -1) //end of file
        currentChar = '\u0000';
      else
        currentChar = (char)i;
    }
    catch(IOException e){
        System.out.println(e);
    }
  }

  private void takeIt(){
    currentSpelling.append(currentChar);
    try{
      int i = inFile.read();
      if(i == -1) //end of file
        currentChar = '\u0000';
      else
        currentChar = (char)i;
    }
    catch(IOException e){
        System.out.println(e);
    }
  }

  private void discard(){
    try{
      int i = inFile.read();
      if(i == -1) //end of file
        currentChar = '\u0000';
      else
        currentChar = (char)i;
    }
```

```
    catch(IOException e){
        System.out.println(e);
    }
  }

  private boolean isDigit(char c){
    return '0' <= c && c <= '9';
  }

  private boolean isLetter(char c){
    return ('a' <= c && c <= 'z') || ('A' <= c && c <= 'Z');
  }

  private boolean isGraphic(char c){
    return c!='"' && c!='‟' && c!='”' && (c == '\t' ||(' ' <= c && c <= '~'));
  }

  private boolean isBlank(char c){
      return (c == ' ' || c == '\n' || c == '\r' || c == '\t');
  }

  private byte scanToken(){
//System.out.println("scanToken: "+currentChar);
    switch(currentChar){

        case 'a': case 'b': case 'c': case 'd': case 'e': case 'f': case 'g':
case 'h': case 'i':
      case 'j': case 'k': case 'l': case 'm': case 'n': case 'o': case 'p': case
'q': case 'r':
      case 's': case 't': case 'u': case 'v': case 'w': case 'x': case 'y': case
'z':
      case 'A': case 'B': case 'C': case 'D': case 'E': case 'F': case 'G': case
'H': case 'I':
      case 'J': case 'K': case 'L': case 'M': case 'N': case 'O': case 'P': case
'Q': case 'R':
      case 'S': case 'T': case 'U': case 'V': case 'W': case 'X': case 'Y': case
'Z':
        takeIt();
        while(isLetter(currentChar) || isDigit(currentChar))
          takeIt();
        return Token.NONTERMINAL;

        case '"': case '‟': case '”':
            discard();
            while(isGraphic(currentChar))
              takeIt();
            if (currentChar=='"' || currentChar=='‟' || currentChar=='”')
                discard();
            return Token.TERMINAL;
        case ':':
            takeIt();
            if(currentChar != ':'){
                new Error("wrong token [" + currentChar+"]", line);
                return Token.EOT;
            }
            takeIt();
            if(currentChar != '='){
```

```
                new Error("wrong token [" + currentChar+"]", line);
                return Token.EOT;
            }
            takeIt();
            return Token.RULE;
        case '.':
            takeIt();
            return Token.ENDRULE;
        case '|':
            takeIt();
            return Token.ALTERNATIVE;
        case '\u0000':
            return Token.EOT;
      default:
        new Error("wrong token [" + currentChar+"]", line);
        return Token.EOT;
    }



}

private void scanSeparator(){
  switch(currentChar){
    case '!':
      discard();
      while(isGraphic(currentChar))
        discard();
      if(currentChar == '\r')
        discard();
      discard();
      line++;
      break;
    case ' ': case '\n': case '\r': case '\t':
      if(currentChar == '\n')
        line++;
      discard();
      while(isBlank(currentChar)){
          if(currentChar == '\n')
            line++;
          discard();
      }
  }
}

public Token scan(){
  currentSpelling = new StringBuffer("");
  while(currentChar == '!' || currentChar == ' ' || currentChar == '\n' ||
        currentChar == '\r' || currentChar == '\t')
    scanSeparator();
  currentKind = scanToken();
  return new Token(currentKind, currentSpelling.toString(), line);
}



public static void main(String[] args){
```

```
    SourceFile sourceFile = new SourceFile();
    Token token;
    Scanner s = new Scanner(sourceFile.openFile());
    do{
      token = s.scan();
      System.out.println("Line: " + token.line + ", spelling = [" +
        token.spelling + "], " + "kind = " + token.kind);
    }while(token.kind != Token.EOT);
  }
}
```

astscanner.SourceFile.java

```
package astscanner;
import java.io.*;
public class SourceFile{
  public BufferedReader openFile(){
    String fileName="";
    BufferedReader inFile=null;
    BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
//    System.out.print("Source file = ");
    System.out.flush();
    try{
//      fileName = stdin.readLine();
//      inFile = new BufferedReader(new FileReader(fileName));

//          inFile = new BufferedReader(new FileReader("C:\\Documents and
Settings\\jfoure\\Desktop\\cs605\\Thesis\\test.txt"));
//          inFile = new BufferedReader(new FileReader("C:\\Documents and
Settings\\jfoure\\Desktop\\cs605\\Thesis\\test2.txt"));
//          inFile = new BufferedReader(new FileReader("C:\\Documents and
Settings\\jfoure\\Desktop\\cs605\\Thesis\\test3.txt"));
            inFile = new BufferedReader(new FileReader("C:\\Documents and
Settings\\jfoure\\Desktop\\cs605\\Thesis\\astinput.txt"));


    }
    catch(FileNotFoundException e){
      System.out.println("The source file " + fileName + " was not found.");
    }

    return inFile;
  }
}
```

astscanner.Token.java

```
package astscanner;
public class Token{
  public byte kind;
  public String spelling;
  public int line;
```

```
  public String toString(){
      return "Token ["+kind+", "+spelling+", "+line+"]";
  }

  public Token(byte kind, String spelling, int line){
    this.kind = kind;
    this.spelling = spelling;
    this.line = line;

/*
    if(kind == TERMINAL)
      for(int k = RULETERMINAL; k <= CURLYBRACKETCLOSE; k++)
        if(spelling.equals(spellings[k])){
          this.kind = (byte)k;
          break;
        }
*/

  }

  public final static byte
      TERMINAL    = 0,        //something that has no production rule
    NONTERMINAL = 1,          //has a production rule
      RULE        = 2,        // ::=
      ENDRULE         = 3,          // .
      ALTERNATIVE = 4,        // |
      EOT         = 5;


  private final static String[] spellings = {

"<terminal>","<nonterminal>","<rule>","<endrule>","<alternative>","<quote>","<eo
t>",
    "::=",".","|","(",")","[","]","{","}"
  };

}
```

astsets.StarterSet.java

```
/*
 * Created on Feb 15, 2004
 */
package astsets;
import astast.*;
import java.util.*;
/**
 * @author jfoure
 */
public class StarterSet{
      public Set symbolSet = new HashSet();
      boolean canBeEmpty = false;

      /**
       * @return
```

```
	 */
	public boolean isCanBeEmpty() {
		return canBeEmpty;
	}

	/**
	 * @return
	 */
	public Set getSymbolSet() {
		return symbolSet;
	}

	public Set getSymbolSpellingSet() {
		Set symbolSpellingSet = new HashSet();
		Iterator iterator = getSymbolSet().iterator();
			while (iterator.hasNext()){
				Symbol symbol = (Symbol) iterator.next();
				if (symbol instanceof NonTerminalSymbol)

	symbolSpellingSet.add(((NonTerminalSymbol)symbol).nonTerminal.spelling);
				if (symbol instanceof TerminalSymbol)

	symbolSpellingSet.add("\""+((TerminalSymbol)symbol).terminal.spelling+"\""
);
			}
			return	symbolSpellingSet;
	}

	/**
	 * @param b
	 */
	public void setCanBeEmpty(boolean b) {
		canBeEmpty = b;
	}

	/**
	 * @param set
	 */
	public void setSymbolSet(Set set) {
		symbolSet = set;
	}

	public void add(Object object){
		symbolSet.add(object);
	}

	public void addAll(Collection collection){
		symbolSet.addAll(collection);
	}
	public void addAll(StarterSet starterSet){
		symbolSet.addAll(starterSet.getSymbolSet());
		if (starterSet.isCanBeEmpty())
			this.setCanBeEmpty(true);
	}
```

```
	public String toString(){
		StringBuffer out = new StringBuffer();
		out.append("StarterSet [ ");
		out.append(canBeEmpty);
		out.append(":");

		Iterator iterator = getSymbolSet().iterator();
		while (iterator.hasNext()){
			Symbol symbol = (Symbol) iterator.next();
			if (symbol instanceof NonTerminalSymbol)

	out.append("**"+((NonTerminalSymbol)symbol).nonTerminal.spelling+"**");
			if (symbol instanceof TerminalSymbol)
				out.append(((TerminalSymbol)symbol).terminal.spelling);
			if (iterator.hasNext())
				out.append(", ");
		}
		out.append("]");
		return out.toString();
	}
}
```

astsets.StarterSetChecker.java

```
package astsets;
import astast.*;
import ebnfscanner.*;
import ebnfparser.*;
import common.*;

import java.util.*;

public class StarterSetChecker implements Visitor{
	//store all production rules to access them backwards List
productionRuleList = new ArrayList();

	public StarterSetChecker(){
	}

	public void check(Grammar grammar){
		grammar.visit(this, null);
	}
	public Object visitGrammar(Grammar grammar, Object arg, int line){
		grammar.productionRuleList.visit(this,null); return null;
	}
	public Object visitProductionRuleList(ProductionRuleList
productionrulelist, Object arg, int line){
		productionrulelist.productionRule.visit(this,null); if
(productionrulelist.productionRuleListTail!=null)
			productionrulelist.productionRuleListTail.visit(this,null);
		return null;
	}
	public Object visitProductionRuleListTail(ProductionRuleListTail
productionrulelisttail, Object arg, int line){
```

```
            productionrulelisttail.productionRule.visit(this,null); if
(productionrulelisttail.productionRuleListTail!=null)

      productionrulelisttail.productionRuleListTail.visit(this,null);
            return null;
      }
      public Object visitProductionRule(ProductionRule productionrule, Object
arg, int line){
            productionRuleList.add(productionrule);
            productionrule.nonTerminal.visit(this,null);
productionrule.definition.visit(this,null); return null;
      }
      public Object visitDefinition(Definition definition, Object arg, int
line){
            definition.symbol.visit(this,null); if
(definition.alternativeDefinition!=null)
                  definition.alternativeDefinition.visit(this,null);
            if (definition.sequenceDefinition!=null)
                  definition.sequenceDefinition.visit(this,null);
            return null;
      }
      public Object visitAlternativeDefinition(AlternativeDefinition
alternativedefinition, Object arg, int line){
            alternativedefinition.symbol.visit(this,null); if
(alternativedefinition.alternativeDefinition!=null)
                  alternativedefinition.alternativeDefinition.visit(this,null);
            return null;
      }
      public Object visitSequenceDefinition(SequenceDefinition
sequencedefinition, Object arg, int line){
            sequencedefinition.symbol.visit(this,null); if
(sequencedefinition.sequenceDefinition!=null)
                  sequencedefinition.sequenceDefinition.visit(this,null);
            return null;
      }
      public Object visitTerminalSymbol(TerminalSymbol terminalsymbol, Object
arg, int line){
            terminalsymbol.terminal.visit(this,null);
            return null;
      }
      public Object visitTerminal(Terminal terminal, Object arg, int line){
            return null;
      }
      public Object visitNonTerminalSymbol(NonTerminalSymbol nonterminalsymbol,
Object arg, int line){
            nonterminalsymbol.nonTerminal.visit(this,null);
            return null;
      }
      public Object visitNonTerminal(NonTerminal nonterminal, Object arg, int
line){
            return null;
      }



      public static void main(String[] args) throws Exception{
```

```
            Parser parser = new Parser();
            ebnfast.Grammar ebnfGrammar = parser.parse();
            //new Display(grammar);
            ebnf2ast.EBNF2ASTConverter ebnf2ASTConverter = new
ebnf2ast.EBNF2ASTConverter(new emt.Token());
            ebnf2ASTConverter.check(ebnfGrammar);
            Grammar astGrammar = ebnf2ASTConverter.getAstGrammar();

            StarterSetChecker check = new StarterSetChecker();
            check.check(astGrammar);
            System.out.println("*****");
            StarterSetTable starterSetTable = check.findStarterSet();
            Iterator starterSetListIterator =
starterSetTable.getStarterSetEntryList().iterator();
            while (starterSetListIterator.hasNext()){
                  StarterSetEntry starterSetEntry = (StarterSetEntry)
starterSetListIterator.next();
                  System.out.println(starterSetEntry);
            }

            System.out.println("The program is terminated normally.");

      }

      public StarterSetTable findStarterSet(){
            //store starter ebnfsets after they are done
            StarterSetTable starterSetTable = new StarterSetTable();
//          Iterator productionRuleListIterator = productionRuleList.iterator();
//          while (productionRuleListIterator.hasNext()){
//                ProductionRule productionRule = (ProductionRule)
productionRuleListIterator.next();
            for (int i=productionRuleList.size()-1;i>=0;i--){
                  ProductionRule productionRule = (ProductionRule)
productionRuleList.get(i);
                  StarterSet starterSet =
findStarterSet(productionRule.definition,starterSetTable);

                  StarterSetEntry starterSetEntry = new
StarterSetEntry(productionRule.nonTerminal.spelling,starterSet);
                  starterSetTable.addStarterSetEntry(starterSetEntry);

                  //System.out.println(productionRule.nonTerminal.spelling+"==>
"+starterSet);
                  System.out.println(starterSetEntry);

            }

            int passcount = starterSetTable.getStarterSetEntryList().size();
            //remove remaining nonTerminals
            while (starterSetTable.containsUnresolvedStarterSetEntries() &&
passcount-->0){
                  Iterator starterSetListIterator =
starterSetTable.getStarterSetEntryList().iterator();
                  while (starterSetListIterator.hasNext()){
                        StarterSetEntry starterSetEntry = (StarterSetEntry)
starterSetListIterator.next();
                        //get keys for elements that are defined
```

```
                        Iterator starterSetMapKeysIterator =
starterSetTable.getNonTerminalNames().iterator();
                        while (starterSetMapKeysIterator.hasNext()){
                              String nonTerminalName = (String)
starterSetMapKeysIterator.next();
                              if
(starterSetEntry.containsNonTerminal(nonTerminalName)){
                                    System.out.println("*#*#*#Found
match:"+nonTerminalName);

      starterSetEntry.removeNonTerminal(nonTerminalName);

      starterSetEntry.getStarterSet().addAll(starterSetTable.getStarterSet(nonTe
rminalName));

                              }
                        }

                  }
            }

            if (passcount==-1)
                  System.out.println("Ended loop without finding all starter
sets.  All remaining non-terminals are tokens.");
            return starterSetTable;
      }

      public static StarterSet findStarterSet(Definition
definition,StarterSetTable starterSetTable){
            if (definition==null)
                  return null;
            StarterSet starterSet = new StarterSet();
            StarterSet childStarterSet =
findStarterSet(definition.symbol,starterSetTable);
            starterSet.addAll(childStarterSet.getSymbolSet());
            if (childStarterSet.isCanBeEmpty()){
                  starterSet.setCanBeEmpty(true);
            }

            if (definition.sequenceDefinition!=null){
                  SequenceDefinition currentSequenceDefinition =
definition.sequenceDefinition;
                              while (currentSequenceDefinition!=null &&
starterSet.isCanBeEmpty()){
                                    Symbol symbol =
currentSequenceDefinition.symbol;
                                    childStarterSet =
findStarterSet(symbol,starterSetTable);

      starterSet.addAll(childStarterSet.getSymbolSet());
                                    if (!childStarterSet.isCanBeEmpty()){
                                          starterSet.setCanBeEmpty(false);
                                    }

                                    currentSequenceDefinition =
currentSequenceDefinition.sequenceDefinition;
                              }
```

```
            }

            if (definition.alternativeDefinition!=null){
                  AlternativeDefinition currentAlternativeDefinition =
definition.alternativeDefinition;
                  while (currentAlternativeDefinition!=null){
                        Symbol symbol = currentAlternativeDefinition.symbol;
                        childStarterSet =
findStarterSet(symbol,starterSetTable);
                        starterSet.addAll(childStarterSet.getSymbolSet());
                        if (childStarterSet.isCanBeEmpty()){
                              starterSet.setCanBeEmpty(true);
                        }

                        currentAlternativeDefinition =
currentAlternativeDefinition.alternativeDefinition;
                  }

            }


            return starterSet;
      }


      public static StarterSet findStarterSet(Symbol symbol,StarterSetTable
starterSetTable){
            StarterSet starterSet = new StarterSet();
                  if (symbol instanceof TerminalSymbol){
                        TerminalSymbol terminalSymbol = (TerminalSymbol) symbol;
                        starterSet.add(terminalSymbol);
                  }
                  if (symbol instanceof NonTerminalSymbol){
                        NonTerminalSymbol nonTerminalSymbol =
(NonTerminalSymbol) symbol;
                        StarterSet nonTerminalStarterSet =
starterSetTable.getStarterSet(nonTerminalSymbol.nonTerminal.spelling);
                        if (nonTerminalStarterSet!=null){
                              starterSet.addAll(nonTerminalStarterSet);

                        }
                        else{
                              starterSet.add(nonTerminalSymbol);
                        }

                        if (nonTerminalStarterSet!=null &&
nonTerminalStarterSet.isCanBeEmpty())
                              starterSet.setCanBeEmpty(true);
                  }

            if (symbol.canBeEmpty())
                  starterSet.setCanBeEmpty(true);
            return starterSet;
      }
```

```
}
```

astsets.StarterSetEntry.java

```
/*
 * Created on Feb 9, 2004
 */
package astsets;
import java.util.*;
import astast.*;
/**
 * @author jfoure
 */
public class StarterSetEntry {
	private String nonTerminalName = null;
	private StarterSet starterSet = new StarterSet();
	public StarterSetEntry(String nonTerminalName, StarterSet starterSet){
		setNonTerminalName(nonTerminalName);
		setStarterSet(starterSet);
	}


	/**
	 * @return
	 */
	public String getNonTerminalName() {
		return nonTerminalName;
	}

	/**
	 * @return
	 */
	public StarterSet getStarterSet() {
		return starterSet;
	}

	/**
	 * @param string
	 */
	public void setNonTerminalName(String string) {
		nonTerminalName = string;
	}

	/**
	 * @param set
	 */
	public void setStarterSet(StarterSet set) {
		starterSet = set;
	}

	public boolean containsNonTerminal(){
		Iterator iterator = starterSet.getSymbolSet().iterator();
		while (iterator.hasNext()){
			Symbol symbol = (Symbol) iterator.next();
```

```
                if (symbol instanceof NonTerminalSymbol)
                      return true;
          }

          return false;
    }

    public boolean containsNonTerminal(String nonTerminalName){
          Iterator iterator = starterSet.getSymbolSet().iterator();
          while (iterator.hasNext()){
                Symbol symbol = (Symbol) iterator.next();
                if (symbol instanceof NonTerminalSymbol)
                {
                      if
(((NonTerminalSymbol)symbol).nonTerminal.spelling.equals(nonTerminalName))
                            return true;
                }
          }

          return false;
    }

    public boolean removeNonTerminal(String nonTerminalName){
          NonTerminalSymbol nonTerminalSymbol = null;
          Iterator iterator = starterSet.getSymbolSet().iterator();
          while (iterator.hasNext()){
                Symbol symbol = (Symbol) iterator.next();
                if (symbol instanceof NonTerminalSymbol)
                {
                      if
(((NonTerminalSymbol)symbol).nonTerminal.spelling.equals(nonTerminalName))
                      nonTerminalSymbol = (NonTerminalSymbol)symbol;
                }
          }

          if (nonTerminalSymbol!=null)
                return starterSet.getSymbolSet().remove(nonTerminalSymbol);
          return false;
    }



    public String toString(){
          StringBuffer out = new StringBuffer();
          out.append("StarterSetEntry [ ");
          out.append(nonTerminalName);
          out.append(":");
          out.append(starterSet.canBeEmpty);
          out.append(":");
          out.append(containsNonTerminal());
          out.append(":");

          Iterator iterator = starterSet.getSymbolSet().iterator();
          while (iterator.hasNext()){
                Symbol symbol = (Symbol) iterator.next();
                if (symbol instanceof NonTerminalSymbol)
```

```
	out.append("**"+((NonTerminalSymbol)symbol).nonTerminal.spelling+"**");
			if (symbol instanceof TerminalSymbol)
				out.append(((TerminalSymbol)symbol).terminal.spelling);
			if (iterator.hasNext())
				out.append(", ");
		}
		out.append("]");
		return out.toString();
	}
}
```

astsets.StarterSetTable.java

```
/*
 * Created on Feb 27, 2004
 */
package astsets;
import java.util.*;
/**
 * @author jfoure
 */
public class StarterSetTable {
	protected List starterSetEntryList = new ArrayList();
	protected Map starterSetEntryMap = new HashMap();
	public void addStarterSetEntry(StarterSetEntry starterSetEntry){
		starterSetEntryList.add(starterSetEntry);

	starterSetEntryMap.put(starterSetEntry.getNonTerminalName(),starterSetEntr
y);
	}
	public List getStarterSetEntryList(){
		return starterSetEntryList;
	}
	public boolean containsUnresolvedStarterSetEntries(){
		Iterator starterSetEntryListIterator =
starterSetEntryList.iterator();
		while (starterSetEntryListIterator.hasNext()){
			StarterSetEntry starterSetEntry = (StarterSetEntry)
starterSetEntryListIterator.next();
			if (starterSetEntry.containsNonTerminal())
				return true;
		}
		return false;
	}
	public StarterSetEntry getStarterSetEntry(String nonTerminalName){
		return (StarterSetEntry) starterSetEntryMap.get(nonTerminalName);
	}
	public StarterSet getStarterSet(String nonTerminalName){
		if (getStarterSetEntry(nonTerminalName)!=null)
			return getStarterSetEntry(nonTerminalName).getStarterSet();
		else
			return null;
	}
	public Set getNonTerminalNames(){
```

```
            return starterSetEntryMap.keySet();
      }

}
```

common.Error.java

```
package common;
public class Error{
  public Error(String message, int line){
    System.out.println("Line " + line + ": " + message);
    System.exit(0);
  }
}
```

common.File2StringBuffer.java

```
/*
 * Created on Aug 7, 2004
 */
package common;
import java.io.*;
import java.util.*;
/**
 * @author jfoure
 */
public class File2StringBuffer {
      public static String encodeJavaString(String s){
            StringBuffer out = new StringBuffer();
            int chr = -1;
            for (int i=0;i<s.length();i++){
                  chr = s.charAt(i);
                  switch (chr){
                        case '\\':out.append("\\\\");break;
                        case '\"':out.append("\\\"");break;
                        case '\t':out.append("\\t");break;
                        default:out.append((char)chr);
                  }
            }
            return out.toString();
      }

      public static void main(String args[]) throws Exception{
            BufferedReader in = new BufferedReader(new FileReader("C:\\Documents
and
Settings\\jfoure\\Desktop\\cs605\\Thesis\\ebnf\\ebnftokengenerator\\JoshScanner.
java"));
            StringBuffer out = new StringBuffer();
            String line = null;
            while ((line=in.readLine())!=null){

      System.out.println("out.append(\""+encodeJavaString(line)+"\\n\");");
            }
```

```
	}

}
```

common.GeneratedFile.java

```
/*
 * Created on Mar 28, 2004
 */
package common;
import java.io.*;
/**
 * @author jfoure
 */
public class GeneratedFile {
	String filename;
	String content;
	String packageName;

	public GeneratedFile(String filename,String packageName,String content){
		this.filename = filename;
		this.content = content;
		this.packageName = packageName;
	}

	/**
	 * @return
	 */
	public String getContent() {
		return content;
	}

	/**
	 * @return
	 */
	public String getFilename() {
		return filename;
	}

	/**
	 * @param string
	 */
	public void setContent(String string) {
		content = string;
	}

	/**
	 * @param string
	 */
	public void setFilename(String string) {
		filename = string;
	}
```

```
	/**
	 * @return
	 */
	public String getPackageName() {
		return packageName;
	}

	/**
	 * @param string
	 */
	public void setPackageName(String string) {
		packageName = string;
	}

	public void writeOutput(){
		writeOutput(this);
	}

	public static void writeOutput(GeneratedFile astFile){

	writeOutput(astFile.getFilename(),astFile.getPackageName(),astFile.getCont
ent());
	}

	public static void writeOutput(String filename,String packageName,String
data){
		try{
			String packageNamePath = packageName;
			while (packageNamePath.indexOf(".")!=-1){
				int location = packageNamePath.indexOf(".");
				packageNamePath =
packageNamePath.substring(0,location)+"\\"+packageNamePath.substring(location+1,
packageNamePath.length());
			}

			FileWriter out = new FileWriter("C:\\Documents and
Settings\\jfoure\\Desktop\\cs605\\Thesis\\ebnf\\"+packageNamePath+"\\"+filename)
;
			out.write(data);
			out.flush();
			out.close();
		} catch (Exception e) {
			e.printStackTrace();
		}

	}

}
```

common.GenericToken.java

```
/*
 * Created on Apr 3, 2004
 */
package common;
```

```
import java.io.*;
import java.util.*;

import ebnftokengenerator.*;

/**
 * @author jfoure
 */
public class GenericToken implements TokenI{
      public static final Map constantSpellingMap = new HashMap(); public static
final Map variableSpellingMap = new HashMap();
      public int kind;
      public String spelling;
      public int line;

      public static List tokenList = null;
      public GenericToken(){

      }

  public GenericToken(int kind, String spelling, int line){
        this.kind = kind;
        this.spelling = spelling;
        this.line = line;
      }
      public String toString(){
            return "Token line: "+line+" kind: "+kind+" spelling: "+spelling;
      }



      public String lookupConstantSpelling(String spelling){
            Iterator tokenListIterator = tokenList.iterator();
            while (tokenListIterator.hasNext()){
                  TokenEntry entry = (TokenEntry) tokenListIterator.next();
                  if (entry.isReservedWord() &&
spelling.equals(((TokenLiteralDefinition)entry.getTokenDefinition()).getRule()))
                        return entry.getVariableName();
                  if (!entry.isReservedWord() &&
spelling.equals(entry.getInitialDefinition()))
                        return entry.getVariableName();
            }
            System.err.println("))lookupConstantSpelling]"+spelling+"/null");
            return null;
      }
      public String lookupVariableSpelling(String spelling){
            Iterator tokenListIterator = tokenList.iterator();
            while (tokenListIterator.hasNext()){
                  TokenEntry entry = (TokenEntry) tokenListIterator.next();
                  if (entry.isReservedWord() &&
spelling.equals(((TokenLiteralDefinition)entry.getTokenDefinition()).getRule()))
                        return "_"+entry.getVariableName().toLowerCase();
                  if (!entry.isReservedWord() &&
spelling.equals(entry.getInitialDefinition()))
                        return "_"+entry.getVariableName().toLowerCase();
            }
            System.err.println("))lookupVariableSpelling]"+spelling+"/null");
```

```
            return null;
      }

/*
      public String lookupConstantSpelling(String spelling){
            if (constantSpellingMap.get(spelling)==null)

      System.err.println("]]]]]]lookupConstantSpelling]"+spelling+"/"+constantSp
ellingMap.get(spelling));
            return (String)constantSpellingMap.get(spelling);
      }
      public String lookupVariableSpelling(String spelling){
            if (variableSpellingMap.get(spelling)==null)

      System.err.println("]]]]]]lookupVariableSpelling]"+spelling+"/"+variableSp
ellingMap.get(spelling));
            return (String)variableSpellingMap.get(spelling);
      }
*/

      public static void main(String args[]) throws Exception{
            /*
            String settings =
                  ", COMMA\r\n"+
                  "; SEMICOLON\r\n"+
                  "CharacterLiteral CHARLITERAL\n"+
                  "array ARRAY";
            */

      }

      /**
       * @return
       */
      public int getKind() {
            return kind;
      }

      /**
       * @return
       */
      public int getLine() {
            return line;
      }

      /**
       * @return
       */
      public String getSpelling() {
            return spelling;
      }

      /**
       * @return
       */
      public static List getTokenList() {
            return tokenList;
```

```
	}

	/**
	 * @param list
	 */
	public static void setTokenList(List list) {
		tokenList = list;
	}

}
```

common.SourceFile.java

```
package common;
import java.io.*;
public class SourceFile{
	public BufferedReader openFile(){
		return openFile("C:\\Documents and
Settings\\jfoure\\Desktop\\cs605\\Thesis\\ebnf\\common\\test.txt");
	}


	public BufferedReader openFile(String fileName){
    BufferedReader inFile=null;
    BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
//    System.out.print("Source file = ");
    System.out.flush();
    try{
//      fileName = stdin.readLine();
	inFile = new BufferedReader(new FileReader(fileName));

//		inFile = new BufferedReader(new FileReader("C:\\Documents and
Settings\\jfoure\\Desktop\\cs605\\Thesis\\test.txt"));
//		inFile = new BufferedReader(new FileReader("C:\\Documents and
Settings\\jfoure\\Desktop\\cs605\\Thesis\\test2.txt"));
//		inFile = new BufferedReader(new FileReader("C:\\Documents and
Settings\\jfoure\\Desktop\\cs605\\Thesis\\test3.txt"));
//		inFile = new BufferedReader(new FileReader("C:\\Documents and
Settings\\jfoure\\Desktop\\cs605\\Thesis\\test4.txt"));

//emt
//  inFile = new BufferedReader(new FileReader("C:\\Documents and
Settings\\jfoure\\Desktop\\cs605\\Thesis\\emt_ebnf.txt"));
//minitriangle
//  inFile = new BufferedReader(new FileReader("C:\\Documents and
Settings\\jfoure\\Desktop\\cs605\\Thesis\\minitriangle_ebnf.txt"));
//    inFile = new BufferedReader(new FileReader("C:\\Documents and
Settings\\jfoure\\Desktop\\cs605\\Thesis\\ebnf\\common\\test.txt"));


    }
    catch(FileNotFoundException e){
      System.out.println("The source file " + fileName + " was not found.");
    }
    return inFile;
```

```
  }
}
```

common.Test.java

```
/*
 * Created on Apr 4, 2004
 */
package common;
import java.io.FileOutputStream;
import java.util.zip.*;
import java.io.*;

/**
 * @author jfoure
 */
public class Test {
      public static void main(String args[]) throws Exception{
            ZipOutputStream out = new ZipOutputStream(new
FileOutputStream("c:\\test.zip"));
            out.putNextEntry(new ZipEntry("Josh.txt"));
            out.write("lalalalalalalallala".getBytes());
            out.putNextEntry(new ZipEntry("/lalal/Josh2.txt"));
            out.write("kjhakfhskdhkshfkjdsh".getBytes());
            InputStream in =
ClassLoader.getSystemResourceAsStream("common.jar");
            out.putNextEntry(new ZipEntry("common.jar"));
            byte buffer[] = new byte[5000];
            while (in.read(buffer)!=-1)
                  out.write(buffer);
            in = ClassLoader.getSystemResourceAsStream("log4j-1.2rc1.jar");
out.putNextEntry(new ZipEntry("log4j1.2.jar"));
            buffer = new byte[5000];
            while (in.read(buffer)!=-1)
                  out.write(buffer);

            out.flush();
            out.close();
      }
}
```

common.Token.java

```
package common;
import common.*;
import java.util.*;

public class Token implements TokenI{
  public int kind;
  public String spelling;
  public int line;

public String toString(){
```

```
      return "Token line: "+line+" kind: "+kind+" spelling: "+spelling;
}

public Token(){
}

public Token(byte kind, String spelling, int line){
    this.kind = kind;
    this.spelling = spelling;
    this.line = line;
    if(kind == IDENTIFIER)
      for(int k = ARRAY; k <= WHILE; k++)
        if(spelling.equals(spellings[k])){
          this.kind = (byte)k;
          break;
        }
  }

  public String lookupConstantSpelling(String spelling){
      String result = (String) constantSpellingMap.get(spelling);
//    if (result==null)
//          throw new RuntimeException("Token is null: ["+spelling+"]");
      return result;
  }
  public String lookupVariableSpelling(String spelling){
      return (String)variableSpellingMap.get(spelling);
  }


  public final static byte
    IDENTIFIER  =  0,
    INTLITERAL  =  1,
    CHARLITERAL =  2,
    ARRAY       =  4,        // array
    BEGIN       =  5,        // begin
    CONST       =  6,        // const
    DO          =  7,        // do
    ELSE        =  8,        // else
    END         =  9,        // end
    IF          = 10,        // if
    IN          = 11,        // in
    LET         = 12,        // let
    OF          = 13,        // of
    PROC        = 14,        // proc
    SKIP        = 15,        // skip
    THEN        = 16,        // then
    VAR         = 17,        // var
    WHILE       = 18,        // while

    COLON       = 19,        // :
    AND         = 20,        // /\
    ASTERISK    = 21,        //*
    BECOMES     = 22,        //:=
    COMMA       = 23,        //,
    DIV         = 24,        // /
    EOT         = 25,        //end of text
    EQUAL       = 26,        //=
```

```
  GREATER     = 27,        //>
  GEQ         = 28,        // >=
  NEQ         = 29,        // \=
  LBRACKET    = 30,        //[
  LPAREN      = 31,        //(
  LESS        = 32,        //<
  LEQ         = 33,        // <=
  MINUS       = 34,        // -
  MOD         = 35,        // //
  NOT         = 36,        // \
  OR          = 37,        // \/
  PLUS        = 38,        // +
  RBRACKET    = 39,        //]
  RPAREN      = 40,        //)
  SEMICOLON   = 41,        //;
  IS          = 42;        // ~


private final static String[] spellings = {
  "<identifier>", "<intliteral>", "<charliteral>", "<operator>",
  "array", "begin", "const", "do", "else", "end", "if", "in", "let", "of",
  "proc", "skip", "then", "var", "while"
};

private final static Map constantSpellingMap = new HashMap();
static {
    constantSpellingMap.put("Identifier","IDENTIFIER");
    constantSpellingMap.put("IntegerLiteral","INTLITERAL");
    constantSpellingMap.put("CharacterLiteral","CHARLITERAL");
    constantSpellingMap.put("array","ARRAY");
    constantSpellingMap.put("begin","BEGIN");
    constantSpellingMap.put("const","CONST");
    constantSpellingMap.put("do","DO");
    constantSpellingMap.put("else","ELSE");
    constantSpellingMap.put("end","END");
    constantSpellingMap.put("if","IF");
    constantSpellingMap.put("in","IN");
    constantSpellingMap.put("let","LET");
    constantSpellingMap.put("of","OF");
    constantSpellingMap.put("proc","PROC");
    constantSpellingMap.put("skip","SKIP");
    constantSpellingMap.put("then","THEN");
    constantSpellingMap.put("var","VAR");
    constantSpellingMap.put("while","WHILE");
    constantSpellingMap.put(":","COLON");
    constantSpellingMap.put("/\\","AND");
    constantSpellingMap.put("*","ASTERISK");
    constantSpellingMap.put(":=","BECOMES");
    constantSpellingMap.put(",","COMMA");
    constantSpellingMap.put("/","DIV");
    constantSpellingMap.put("EOT","EOT");
    constantSpellingMap.put("=","EQUAL");
    constantSpellingMap.put(">","GREATER");
    constantSpellingMap.put(">=","GEQ");
    constantSpellingMap.put("\\=","NEQ");
    constantSpellingMap.put("[","LBRACKET");
    constantSpellingMap.put("(","LPAREN");
```

```
    constantSpellingMap.put("<","LESS");
    constantSpellingMap.put("<=","LEQ");
    constantSpellingMap.put("-","MINUS");
    constantSpellingMap.put("//","MOD");
    constantSpellingMap.put("\\","NOT");
    constantSpellingMap.put("\\/","OR");
    constantSpellingMap.put("+","PLUS");
    constantSpellingMap.put("]","RBRACKET");
    constantSpellingMap.put(")","RPAREN");
    constantSpellingMap.put(";","SEMICOLON");
    constantSpellingMap.put("~","IS");
};

private final static Map variableSpellingMap = new HashMap();
static {
    variableSpellingMap.put("Identifier","identifier");
    variableSpellingMap.put("IntegerLiteral","intliteral");
    variableSpellingMap.put("CharacterLiteral","charliteral");
    variableSpellingMap.put("array","array");
    variableSpellingMap.put("begin","begin");
    variableSpellingMap.put("const","const");
    variableSpellingMap.put("do","do");
    variableSpellingMap.put("else","else");
    variableSpellingMap.put("end","end");
    variableSpellingMap.put("if","if");
    variableSpellingMap.put("in","in");
    variableSpellingMap.put("let","let");
    variableSpellingMap.put("of","of");
    variableSpellingMap.put("proc","proc");
    variableSpellingMap.put("skip","skip");
    variableSpellingMap.put("then","then");
    variableSpellingMap.put("var","var");
    variableSpellingMap.put("while","while");
    variableSpellingMap.put(":","colon");
    variableSpellingMap.put("/\\","and");
    variableSpellingMap.put("*","asterix");
    variableSpellingMap.put(":=","becomes");
    variableSpellingMap.put(",","comma");
    variableSpellingMap.put("/","div");
    variableSpellingMap.put("EOT","eot");
    variableSpellingMap.put("=","equal");
    variableSpellingMap.put(">","greater");
    variableSpellingMap.put(">=","geq");
    variableSpellingMap.put("\\=","neq");
    variableSpellingMap.put("[","leftBracket");
    variableSpellingMap.put("(","leftParen");
    variableSpellingMap.put("<","less");
    variableSpellingMap.put("<=","leq");
    variableSpellingMap.put("-","minus");
    variableSpellingMap.put("//","mod");
    variableSpellingMap.put("\\","not");
    variableSpellingMap.put("\\/","or");
    variableSpellingMap.put("+","plus");
    variableSpellingMap.put("]","rightBracket");
    variableSpellingMap.put(")","rightParen");
    variableSpellingMap.put(";","semicolon");
    variableSpellingMap.put("~","is");
```

```
  };

/**
 * @return
 */
public int getKind() {
      return kind;
}

/**
 * @return
 */
public int getLine() {
      return line;
}

/**
 * @return
 */
public String getSpelling() {
      return spelling;
}

}
```

common.TokenI.java

```
/*
 * Created on Feb 29, 2004
 */
package common;

/**
 * @author jfoure
 */
public interface TokenI {
      public final static byte EOT = -1;
      public String lookupConstantSpelling(String spelling); public String
lookupVariableSpelling(String spelling);
      public int getKind();
      public String getSpelling();
      public int getLine();


}
```

common.TokenRule.java

```
/*
 * Created on May 11, 2004
 */
package common;
```

```
/**
 * @author jfoure
 */
public class TokenRule {
	private String tokenName;
	private String regularExpression;
	private boolean reservedWord;
	private int kind;
	private boolean ignored;

	public TokenRule(String tokenName, String regularExpression, boolean
reservedWord, int kind, boolean ignored){
		this.tokenName = tokenName;
		this.regularExpression = regularExpression;
		this.reservedWord = reservedWord;
		this.kind = kind;
		this.ignored = ignored;
	}

	public GenericToken createToken(String spelling, int line){
		return new GenericToken(kind,spelling,line);
	}

	/**
	 * @return
	 */
	public String getRegularExpression() {
		return regularExpression;
	}

	/**
	 * @return
	 */
	public String getTokenName() {
		return tokenName;
	}

	/**
	 * @param string
	 */
	public void setRegularExpression(String string) {
		regularExpression = string;
	}

	/**
	 * @param string
	 */
	public void setTokenName(String string) {
		tokenName = string;
	}

	/**
	 * @return
	 */
	public int getKind() {
		return kind;
	}
```

```
	/**
	 * @param i
	 */
	public void setKind(int i) {
		kind = i;
	}


	/**
	 * @return
	 */
	public boolean isIgnored() {
		return ignored;
	}

	/**
	 * @param b
	 */
	public void setIgnored(boolean b) {
		ignored = b;
	}

	/**
	 * @return
	 */
	public boolean isReservedWord() {
		return reservedWord;
	}

	/**
	 * @param b
	 */
	public void setReservedWord(boolean b) {
		reservedWord = b;
	}

}
```

defaults.BadiiSettings.java

```
/*
 * Created on Sep 15, 2004
 */
package defaults;

/**
 * @author jfoure
 */
public class BadiiSettings implements GrammarSettings {
    public String getGrammar() {
        StringBuffer out = new StringBuffer();
            out.append("Program ::= {\"Global\" VarDeclaration} FuncDeclaration
{ FuncDeclaration }.\n");
            out.append("VarDeclaration ::= Type Varlist.\n");
```

```
            out.append("Varlist ::= Identifier {\",\" Identifier}.\n");
            out.append("Type ::= (\"Int\" |\"Line\"|\"Point\"|\"Boolean\")
[\"[\"IntegerLiteral\"]\"].\n");
            out.append("FuncDeclaration ::=  Type Identifier \"(\" [Paramlist]
\")\" {VarDeclaration} StmtBlock \"End\".\n");
            out.append("Paramlist ::= Parameter {\",\" Parameter}.\n");
            out.append("Parameter ::= Type Identifier.\n");
            out.append("StmtBlock ::= Stmt {Stmt}.\n");
            out.append("Stmt ::= Expression \";\" | Assignstmt \";\" | IfBlock |
WhileBlock | Forblock  | ReturnStmt \";\" | BuiltinFunc \";\"| ReadStmt \";\" |
WriteStmt \";\".\n");
            out.append("IfBlock ::= \"If\" Expression \"Then\" StmtBlock
[\"else\" StmtBlock] \"IfEnd\".\n");
            out.append("WhileBlock ::= \"While\" Expression \"Do\" StmtBlock
\"WhileEnd\" .\n");
            out.append("Forblock ::= ForblockDefault | ForblockLine.\n");
            out.append("ForblockDefault ::= \"For\" [Assignstmt] \";\"
Expression \";\" [Assignstmt] \"Do\" StmtBlock \"ForEnd\".\n");
            out.append("ForblockLine ::= \"ForLine\" Identifier \"In\"
Identifier \"Do\" StmtBlock \"ForEnd\".\n");
            out.append("Assignstmt ::= \"Let\" Varaccess \"<-\" Expression.\n");
            out.append("Varaccess ::= Identifier [\"[\"Expression\"]\"] |
Identifier \".\" (\"x\"|\"y\"|IntegerLiteral) .\n");
            out.append("ReturnStmt ::= \"Return\" Expression.\n");
            out.append("BuiltinFunc ::= ColorFunc | DrawFunc | AppendFunc.\n");
            out.append("ColorFunc ::= \"Color\" Expression \";\" Expression
\";\" Expression.\n");
            out.append("DrawFunc ::= \"Draw\" Expression.\n");
            out.append("AppendFunc ::= \"Append\" Expression \"To\"
Identifier.\n");
            out.append("ReadStmt ::=  \"Read\" Varaccess.\n");
            out.append("WriteStmt ::= \"Write\" Expression.\n");
            out.append("Expression  ::= PrimaryExpression [PrimaryOperator
PrimaryExpression].\n");
            out.append("PrimaryOperator ::= \"&&\"|\"||\".\n");
            out.append("PrimaryExpression ::= EqualsEqualsExpression
[EqualsEqualsOperator EqualsEqualsExpression].\n");
            out.append("EqualsEqualsOperator ::= \"==\" | \"!=\".\n");
            out.append("EqualsEqualsExpression ::= SimpleExpression
[RelationalOperator SimpleExpression].\n");
            out.append("RelationalOperator ::= \"<\" | \"<=\" | \">\" |
\">=\".\n");
            out.append("SimpleExpression ::= [\"-\"] Term {AddingOperator
Term}.\n");
            out.append("AddingOperator ::= \"+\" | \"-\".\n");
            out.append("Term ::= Factor {MultiplyingOperator Factor}.\n");
            out.append("MultiplyingOperator ::= \"*\" | \"/\" | \"%\".\n");
            out.append("Factor ::= Constant | Varaccess | \"(\" Expression \")\"
| Funccall | \"!\" Factor.\n");
            out.append("Constant ::= IntegerLiteral | BooleanLiteral |
PointLiteral | LineLiteral.\n");
            out.append("BooleanLiteral ::= \"True\"|\"False\".\n");
            out.append("PointLiteral ::= \"<\" SimpleExpression \",\"
SimpleExpression \">\".\n");
            out.append("LineLiteral ::= \"{\" Factor  Factor {Factor}\"}\".\n");
            out.append("Funccall ::= \"Call\" Identifier \"(\" [Arglist]
\")\".\n");
```

```
            out.append("Arglist ::= Expression {\",\" Expression}.\n");
        return out.toString();
    }

    public String getTokens() {
        StringBuffer out = new StringBuffer();
            out.append("#EMT Lexicon\n");
            out.append("#IGNORED TOKENS\n");
            out.append("SEPERATOR:Seperator\n");
            out.append("COMMENT:Comment\n");
            out.append("#TOKENS\n"); out.append("IDENTIFIER:Identifier\n");
out.append("INTEGER:IntegerLiteral\n");
            out.append("GLOBAL:\"Global\"\n");
            out.append("COMMA:\",\"\n");
            out.append("INT:\"Int\"\n");
            out.append("LINE:\"Line\"\n");
            out.append("POINT:\"Point\"\n");
            out.append("BOOLEAN:\"Boolean\"\n");
            out.append("LBRACKET:\"[\"\n");
            out.append("RBRACKET:\"]\"\n");
            out.append("LPARAM:\"(\"\n");
            out.append("RPARAM:\")\"\n");
            out.append("END:\"End\"\n");
            out.append("COMMAN:\",\"\n");
            out.append("SEMICOLON:\";\"\n");
            out.append("IF:\"If\"\n");
            out.append("THEN:\"Then\"\n");
            out.append("ELSE:\"else\"\n");
            out.append("IFEND:\"IfEnd\"\n");
            out.append("WHILE:\"While\"\n");
            out.append("DO:\"Do\"\n");
            out.append("WHILEEND:\"WhileEnd\"\n");
            out.append("FOR:\"For\"\n");
            out.append("FOREND:\"ForEnd\"\n");
            out.append("FORLINE:\"ForLine\"\n");
            out.append("IN:\"In\"\n");
            out.append("LET:\"Let\"\n");
            out.append("POINTER:\"<-\"\n");
            out.append("PERIOD:\".\"\n");
            out.append("CALL:\"Call\"\n");
            out.append("X:\"x\"\n");
            out.append("Y:\"y\"\n");
            out.append("RETURN:\"Return\"\n");
            out.append("COLOR:\"Color\"\n");
            out.append("DRAW:\"Draw\"\n");
            out.append("APPEND:\"Append\"\n");
            out.append("TO:\"To\"\n");
            out.append("READ:\"Read\"\n");
            out.append("WRITE:\"Write\"\n");
            out.append("AND:\"&&\"\n");
            out.append("OR:\"||\"\n");
            out.append("EQUALS:\"==\"\n");
            out.append("NOTEQUALS:\"!=\"\n");
            out.append("LESSTHAN:\"<\"\n");
            out.append("LESSTHANEQUALS:\"<=\"\n");
            out.append("GREATERTHAN:\">\"\n");
            out.append("GREATERTHANEQUALS:\">=\"\n");
```

```
            out.append("MINUS:\"-\"\n");
            out.append("PLUS:\"+\"\n");
            out.append("MULT:\"*\"\n");
            out.append("DIV:\"/\"\n");
            out.append("MOD:\"%\"\n");
            out.append("NOT:\"!\"\n");
            out.append("TRUE:\"True\"\n");
            out.append("FALSE:\"False\"\n");
            out.append("LCURLY:\"{\"\n");
            out.append("RCURLY:\"}\"\n");

            out.append("#VARIABLES\n");
            out.append("Identifier:[a-z]([a-zA-Z]|\\d)*\n");
            out.append("IntegerLiteral:\\d+\n");
            out.append("Seperator:\\x20|\\t\n");
            out.append("Comment:\\x23([a-zA-Z]|\\d|\\x20)*\n");
        return out.toString();
    }

    public String getSourceCode() {
        StringBuffer out = new StringBuffer();
            out.append("Global Int i\n"); out.append("Line l() 30==20; End\n");
        return out.toString();
    }

    public String getASTPackage() {
        return "badiiast";
    }
    public String getParserPackage() {
        return "badiiparser";
    }

}
```

defaults.EMTSettings.java

```
/*
 * Created on Sep 15, 2004
 */
package defaults;

/**
 * @author jfoure
 */
public class EMTSettings implements GrammarSettings {
    public String getGrammar() {
        StringBuffer out = new StringBuffer();
        out.append("Program ::= SingleCommand .\n");
        out.append("Command ::= SingleCommand {\";\" SingleCommand}.\n");
        out.append(
            "SingleCommand ::= Identifier AssignOrCallCommandTail | \"if\"
Expression \"then\" SingleCommand \"else\" SingleCommand | \"while\" Expression
\"do\" SingleCommand | \"let\" Declaration \"in\" SingleCommand | \"begin\"
Command \"end\" | \"skip\".\n");
```

```
        out.append("AssignOrCallCommandTail ::= { \"[\" Expression \"]\" }
\":=\" Expression | \"(\" [ActualParameterSequence] \")\".\n");
        out.append("Expression ::= PrimaryExpression {PrimaryOperator
PrimaryExpression}.\n");
        out.append("PrimaryOperator ::= \"/\\\" | \"\\/\".\n");
        out.append("PrimaryExpression ::= SimpleExpression [RelationalOperator
SimpleExpression].\n");
        out.append("RelationalOperator ::= \"<\" | \"<=\" | \">\" | \">=\" |
\"=\" | \"\\=\".\n");
        out.append("SimpleExpression ::= [\"-\"] Term {AddingOperator
Term}.\n");
        out.append("AddingOperator ::= \"+\" | \"-\".\n");
        out.append("Term ::= Factor {MultiplyingOperator Factor}.\n");
        out.append("MultiplyingOperator ::= \"*\" | \"/\" | \"//\".\n");
        out.append("Factor ::= IntegerLiteral | CharacterLiteral|Vname | \"(\"
Expression \")\" | \"\\\" Factor.\n");
        out.append("Vname ::= Identifier {\"[\"Expression\"]\"}.\n");
        out.append("ActualParameterSequence ::= ActualParameter [\",\"
ActualParameterSequence].\n");
        out.append("ActualParameter ::= Expression | \"var\" Vname.\n");
        out.append("Declaration ::= singleDeclaration {\";\" singleDeclaration
}.\n");
        out.append(
            "singleDeclaration ::= \"const\" Identifier \"~\" Constant |\"var\"
IdentifierList \":\" Typedenoter | \"proc\" Identifier
\"(\"[FormalParameterSequence]\")\" \"~\" SingleCommand.\n");
        out.append("Constant ::= IntegerLiteral | CharacterLiteral |
Identifier.\n");
        out.append("IdentifierList ::= Identifier {\",\" Identifier}.\n");
        out.append("Typedenoter ::= Identifier | \"array\" Constant \"of\"
Typedenoter.\n");
        out.append("FormalParameterSequence ::= FormalParameter [\",\"
FormalParameterSequence].\n");
        out.append("FormalParameter ::= Identifier \":\" Typedenoter | \"var\"
Identifier \":\" Typedenoter.");
        return out.toString();
    }

    public String getTokens() {
        StringBuffer out = new StringBuffer();
        out.append("#EMT Lexicon\n");
        out.append("#IGNORED TOKENS\n");
        out.append("SEPERATOR:Seperator\n");
        out.append("COMMENT:Comment\n");
        out.append("\n");
        out.append("#TOKENS\n");
        out.append("\n");
        out.append("IDENTIFIER:Identifier\n");
        out.append("INTEGERLITERAL:IntegerLiteral\n");
        out.append("CHARACTERLITERAL:CharacterLiteral\n");
        out.append("ARRAY:\"array\"\n");
        out.append("BEGIN:\"begin\"\n");
        out.append("CONST:\"const\"\n");
        out.append("DO:\"do\"\n");
        out.append("ELSE:\"else\"\n");
        out.append("END:\"end\"\n");
        out.append("IF:\"if\"\n");
```

```
out.append("IN:\"in\"\n");
out.append("LET:\"let\"\n");
out.append("OF:\"of\"\n");
out.append("PROC:\"proc\"\n");
out.append("SKIP:\"skip\"\n");
out.append("THEN:\"then\"\n");
out.append("VAR:\"var\"\n");
out.append("WHILE:\"while\"\n");
out.append("COLON:\":\"\n");
out.append("SEMICOLON:\";\"\n");
out.append("COMMA:\",\"\n");
out.append("BECOMES:\":=\"\n");
out.append("NOT:\"\\\"\n");
out.append("LPAREN:\"(\"\n");
out.append("RPAREN:\")\"\n");
out.append("LBRACKET:\"[\"\n");
out.append("RBRACKET:\"]\"\n");
out.append("AND:\"/\\\"\n");
out.append("ASTERISK:\"*\"\n");
out.append("DIV:\"/\"\n");
out.append("EQUAL:\"=\"\n");
out.append("GREATER:\">\"\n");
out.append("GEQ:\">=\"\n");
out.append("NEQ:\"\\=\"\n");
out.append("LESS:\"<\"\n");
out.append("LEQ:\"<=\"\n");
out.append("MINUS:\"-\"\n");
out.append("MOD:\"//\"\n");
out.append("OR:\"\\/\"\n");
out.append("PLUS:\"+\"\n");
out.append("IS:\"~\"\n");
/*
out.append("LPAREN:\"\x28\"\n");
out.append("RPAREN:\"\x29\"\n");
out.append("LBRACKET:\"\x5B\"\n");
out.append("RBRACKET:\"\x5D\"\n");
out.append("AND:\"\x2F\\\"\n");
out.append("ASTERISK:\"\x2A\"\n");
out.append("DIV:\"\x2F\"\n");
out.append("EQUAL:\"\x3D\"\n");
out.append("GREATER:\"\x3E\"\n");
out.append("GEQ:\"\x3E\x3D\"\n");
out.append("NEQ:\"\\\x3D\"\n");
out.append("LESS:\"\x3C\"\n");
out.append("LEQ:\"\x3C\x3D\"\n");
out.append("MINUS:\"\x2D\"\n");
out.append("MOD:\"\x2F\x2F\"\n");
out.append("OR:\"\\\x2F\"\n");
out.append("PLUS:\"\x2B\"\n");
out.append("IS:\"\x7E\"\n");
out.append("BLAH:\"aa\x7E\"\n");
*/
out.append("\n");
out.append("#VARIABLES\n");
out.append("IntegerLiteral:\\d+\n");
out.append("CharacterLiteral:\'\\w\'\n");
out.append("Identifier:[a-zA-Z]([a-zA-Z]|\\d)*\n");
```

```
        out.append("Comment:!(\\S? ?)*\n");
        out.append("Seperator:\\x20|\\t\n");

        return out.toString();
    }

    public String getSourceCode() {
        StringBuffer out = new StringBuffer();
        out.append("!This is a comment.It continues to the end - of - line.\n");
        out.append("let \n");
        out.append(" const m ~8;\n");
        out.append(" const j ~8 \n");
        out.append("!var n : integer;\n");
        out.append("!var x, y, z : array 5 of array 6 of integer;\n");
        out.append("!proc p() ~let var x : char in begin x : = y;p() end;\n");
        out.append("!proc q(var x : array 5 of Boolean, y : integer) ~skip \n");
        out.append("in \n");
        out.append(" begin \n");
        out.append(" if (- true) then skip else skip;\n");
        out.append(" if (true) then skip else skip;\n");
        out.append("!x[m + 2] : = 2;\n");
        out.append("!n : = 2 * m * m;\n");
        out.append("!putint(n);\n");
        out.append("!f(a[2][3], var a);\n");
        out.append(" x := x < y * 6 \n");
        out.append("end");

        return out.toString();
    }

    public String getASTPackage() {
        return "emtast";
    }
    public String getParserPackage() {
        return "emtparser";
    }

}
```

defaults.GrammarSettings.java

```
/*
 * Created on Sep 15, 2004
 */
package defaults;

/**
 * @author jfoure
 */
public interface GrammarSettings extends java.io.Serializable{
    public abstract String getGrammar();
    public abstract String getTokens();
    public abstract String getSourceCode();
    public abstract String getASTPackage();
    public abstract String getParserPackage();
```

```
}
```

defaults.GrammarSettingsImpl.java

```
/*
 * Created on Sep 15, 2004
 */
package defaults;

/**
 * @author jfoure
 */
public class GrammarSettingsImpl implements GrammarSettings {
      public String grammar = null;
      public String tokens = null;
      public String sourceCode = null;
      public String astPackage = null;
      public String parserPackage = null;


    /**
     * @return
     */
    public String getASTPackage() {
        return astPackage;
    }

    /**
     * @return
     */
    public String getGrammar() {
        return grammar;
    }

    /**
     * @return
     */
    public String getParserPackage() {
        return parserPackage;
    }

    /**
     * @return
     */
    public String getSourceCode() {
        return sourceCode;
    }

    /**
     * @return
     */
    public String getTokens() {
        return tokens;
    }
```

```
    /**
     * @param string
     */
    public void setASTPackage(String string) {
        astPackage = string;
    }

    /**
     * @param string
     */
    public void setGrammar(String string) {
        grammar = string;
    }

    /**
     * @param string
     */
    public void setParserPackage(String string) {
        parserPackage = string;
    }

    /**
     * @param string
     */
    public void setSourceCode(String string) {
        sourceCode = string;
    }

    /**
     * @param string
     */
    public void setTokens(String string) {
        tokens = string;
    }

}
```

defaults.Main.java

```
/*
 * Created on Oct 24, 2004
 */
package defaults;
import java.io.*;
import java.util.*;

/**
 * @author jfoure
 */
public class Main {
      public static void printSettings(GrammarSettings settings){
            System.out.println("Language: "+settings.getClass().getName());
            System.out.println();
            System.out.println("AST Package Name: "+settings.getASTPackage());
```

```
            System.out.println("Scanner/Parser Package Name:
"+settings.getParserPackage());
            System.out.println();
            System.out.println();
            System.out.println("EBNF:");
            System.out.println();
            System.out.println(settings.getGrammar());
            System.out.println();
            System.out.println();
            System.out.println("Token:");
            System.out.println();
            System.out.println(settings.getTokens());
            System.out.println();
            System.out.println();
            System.out.println("Sample Code:");
            System.out.println();
            System.out.println(settings.getSourceCode());
      }

    public static void main(String[] args) {
      Package defaults = Package.getPackage("defaults");
      System.out.println(defaults);
      File dir = new File("defaults");
            System.out.println(dir);
      if (dir.isDirectory()){
                  System.out.println("is dir");
                  File classes[] = dir.listFiles();
                  for (int i=0;i<classes.length;i++){
//                      System.out.println(classes[i]);
                        try {
                              String filename = classes[i].getName();
                              //System.out.println("Filename:"+filename);
                              if (filename.endsWith(".class")){
                                    String classname =
filename.substring(0,filename.length()-6);
                                    classname =
classname.replace(File.separatorChar,'.');
//                                  System.out.println("class:"+classname);
                                    Class c =
Class.forName("defaults."+classname);
                                    Object o = c.newInstance();
                                    if (o instanceof GrammarSettings){
                                          printSettings((GrammarSettings)o);

                                    }
                                    Class interfaces[] = c.getInterfaces();
                                    for (int j=0;j>interfaces.length;j++){
                                          System.out.println("="+interfaces[j]);
                                    }

/*
                                    classname = "defaults."+classname;
                                    Class c =
ClassLoader.getSystemClassLoader().loadClass(classname);
                                    System.out.println("class:"+c);
                                    Class interfaces[] = c.getDeclaredClasses();
                                    for (int j=0;j>interfaces.length;j++){
```

```
                                        System.out.println("="+interfaces[j]);
                                    }
*/

                                }
                            } catch (Exception e){
                                System.out.println(e);
                            }
                        }
                }

//          printSettings(new EMTSettings());

    }
}
```

defaults.PascalSettings.java

```
/*
 * Created on Sep 15, 2004
 */
package defaults;

/**
 * @author jfoure
 */
public class PascalSettings implements GrammarSettings {
    public String getGrammar() {
        StringBuffer out = new StringBuffer();
            out.append("Program ::= \"program\" ProgramName \";\" BlockBody
\".\".\n");
            out.append("BlockBody ::= [ ConstantDefinitionPart ] [
TypeDefinitionPart ]     [ VariableDefinitionPart ] { ProcedureDefinition }
CompoundStatement.\n");
            out.append("ConstantDefinitionPart ::= \"const\" ConstantDefinition
{ ConstantDefinition }.\n");
            out.append("ConstantDefinition ::= Constant \"=\" Constant
\";\".\n");
            out.append("TypeDefinitionPart ::= \"type\" TypeDefinition {
TypeDefinition }.\n");
            out.append("TypeDefinition ::= TypeName \"=\" NewType \";\".\n");
            out.append("NewType ::= NewArrayType | NewRecordType.\n");
            out.append("NewArrayType ::= \"array\" \"[\" IndexRange \"]\" \"of\"
TypeName.\n");
            out.append("IndexRange ::= Constant \"..\" Constant.\n");
            out.append("NewRecordType ::= \"record\" FieldList \"end\".\n");
            out.append("FieldList ::= RecordSection { \";\" RecordSection
}.\n");
            out.append("RecordSection ::= FieldName { \",\" FieldName  } \":\"
TypeName.\n");
            out.append("VariableDefinitionPart ::= \"var\"
VariableDefinitionPart { VariableDefinitionPart }.\n");
            out.append("VariableDefinition ::= VariableGroup \";\".\n");
            out.append("VariableGroup ::= VariableName  { \",\" VariableName }
\":\" TypeName.\n");
```

```
            out.append("ProcedureDefinition ::= \"procedure\" ProcedureName
ProcedureBlock \";\".\n");
            out.append("ProcedureBlock ::= [ \"(\" FormalParameterList \")\" ]
\";\" BlockBody .\n");
            out.append("FormalParameterList ::= ParameterDefinition { \";\"
ParameterDefinition }.\n");
            out.append("ParameterDefinition ::= [\"var\"] VariableGroup.\n");
            out.append("Statement ::= [StatementBody] .\n");
            out.append("StatementBody ::= AssignmentOrProcedureStatement |
IfStatement | WhileStatement | CompoundStatement.\n");
            out.append("AssignmentOrProcedureStatement ::= Name
AssignmentOrProcedureStatementTail.\n");
            out.append("AssignmentOrProcedureStatementTail ::=
AssignmentStatementTail | ProcedureStatementTail.\n");
            out.append("AssignmentStatementTail ::= VariableAccessTail \":=\"
Expression.\n");
            out.append("ProcedureStatementTail ::= [ \"(\" ActualParameterList
\")\" ].\n");
            out.append("ActualParameterList ::= ActualParameterList {\",\"
ActualParameter}.\n");
            out.append("ActualParameter ::= Expression.\n");
            out.append("IfStatement ::= \"if\" Expression \"then\" Statement [
\"else\" Statement ].\n");
            out.append("WhileStatement ::= \"while\" Expression \"do\"
Statement.\n");
            out.append("CompoundStatement ::= \"{\" Statement { \";\" Statement
} \"end\".\n");
            out.append("Expression ::= SimpleExpression [ RelationalOperator
SimpleExpression ].\n");
            out.append("RelationalOperator ::= \"<\" | \"=\" | \"<>\" |
\">\".\n");
            out.append("SimpleExpression ::= [ SignOperator ] Term {
AddingOperator Term }.\n");
            out.append("SignOperator ::= \"+\" | \"-\".\n");
            out.append("AddingOperator ::= \"+\" | \"-\" | \"or\".\n");
            out.append("Term ::= Factor { MultiplyOperator Factor }.\n");
            out.append("MultiplyOperator ::= \"*\" | \"div\" | \"mod\" |
\"and\".\n");
            out.append("Factor ::= Constant | VariableAccess | \"(\" Expression
\")\" | \"not\" Factor.\n");
            out.append("VariableAccess ::= VariableName VariableAccessTail.\n");
            out.append("VariableAccessTail ::= {Selector}.\n");
            out.append("Selector ::= IndexSelector | FieldSelector.\n");
            out.append("IndexSelector ::= \"[\" Expression \"]\".\n");
            out.append("FieldSelector ::= \".\" FieldName.\n");
            out.append("Constant ::= Numeral | ConstantName.\n");
            out.append("ProgramName ::= Identifier.\n");
            out.append("TypeName ::= Identifier.\n");
            out.append("FieldName ::= Identifier.\n");
            out.append("VariableName ::= Identifier.\n");
            out.append("ProcedureName ::= Identifier.\n");
            out.append("Name ::= Identifier.\n");
            out.append("ConstantName ::= ConstantIdentifier.\n");

        return out.toString();
    }
```

```
public String getTokens() {
    StringBuffer out = new StringBuffer();
        out.append("#EMT Lexicon\n");
        out.append("#IGNORED TOKENS\n");
        out.append("SEPERATOR:Seperator\n");
        out.append("\n");
        out.append("#TOKENS\n");
        out.append("\n");
        out.append("EMPTY:Empty\n");
        out.append("NUMERAL:Numeral\n");
        out.append("IDENTIFIER:Identifier\n");
        out.append("CONSTANTIDENTIFIER:ConstantIdentifier\n");
        out.append("PROGRAM:\"program\"\n");
        out.append("SEMICOLON:\";\"\n");
        out.append("PERIOD:\".\"\n");
        out.append("CONST:\"const\"\n");
        out.append("EQUALS:\"=\"\n");
        out.append("TYPE:\"type\"\n");
        out.append("ARRAY:\"array\"\n");
        out.append("LBRACKET:\"[\"\n");
        out.append("RBRACKET:\"]\"\n");
        out.append("OF:\"of\"\n");
        out.append("RANGE:\"..\"\n");
        out.append("RECORD:\"record\"\n");
        out.append("END:\"end\"\n");
        out.append("COMMA:\",\"\n");
        out.append("COLON:\":\"\n");
        out.append("VAR:\"var\"\n");
        out.append("PROCEDURE:\"procedure\"\n");
        out.append("LPAREN:\"(\"\n");
        out.append("RPAREN:\")\"\n");
        out.append("BECOMES:\":=\"\n");
        out.append("IF:\"if\"\n");
        out.append("THEN:\"then\"\n");
        out.append("ELSE:\"else\"\n");
        out.append("WHILE:\"while\"\n");
        out.append("DO:\"do\"\n");
        out.append("LCURLYBRACKET:\"{\"\n");
        out.append("LESSTHAN:\"<\"\n");
        out.append("GREATERTHAN:\">\"\n");
        out.append("NOTEQUALS:\"<>\"\n");
        out.append("PLUS:\"+\"\n");
        out.append("MINUS:\"-\"\n");
        out.append("OR:\"or\"\n");
        out.append("MULT:\"*\"\n");
        out.append("DIV:\"div\"\n");
        out.append("MOD:\"mod\"\n");
        out.append("AND:\"and\"\n");
        out.append("NOT:\"not\"\n");
        out.append("\n");
        out.append("#VARIABLES\n");
        out.append("Identifier:[a-z]([a-zA-Z]|\\d)*\n");
        out.append("ConstantIdentifier:[A-Z]([A-Z]|\\d)*\n");
        out.append("Numeral:\\d+\n");
        out.append("Empty:\n");
        out.append("Seperator:\\x20|\\t\n");
```

```
        return out.toString();
    }

    public String getSourceCode() {
        StringBuffer out = new StringBuffer();
            out.append("program josh;\n"); out.append("const LALA = BLAH;\n");
out.append("{ while LALA < BLAH do b := 2\n"); out.append("end.");
        return out.toString();
    }

    public String getASTPackage() {
        return "pascalast";
    }
    public String getParserPackage() {
        return "pascalparser";
    }

}
```

defaults.PLSettings.java

```
/*
 * Created on Sep 15, 2004
 */
package defaults;

/**
 * @author jfoure
 */
public class PLSettings implements GrammarSettings {
    public String getGrammar() {
        StringBuffer out = new StringBuffer();
            out.append("Program ::= Block \".\".\n");
            out.append("Block ::= \"begin\" DefinitionPart StatementPart
\"end\".\n");
            out.append("DefinitionPart ::= {Definition \";\"}.\n");
            out.append("Definition ::= ConstantDefinition | VariableDefinition |
ProcedureDefinition.\n");
            out.append("ConstantDefinition ::= \"const\" ConstantName \"=\"
Constant.\n");
            out.append("VariableDefinition ::= TypeSymbol
VariableDefinitionTail.\n");
            out.append("VariableDefinitionTail ::= VariableList | \"array\"
VariableList \"[\" Constant \"]\".\n");
            out.append("TypeSymbol ::= \"integer\" | \"boolean\".\n");
            out.append("VariableList ::= VariableName {\",\" VariableName}.\n");
            out.append("ProcedureDefinition ::= \"proc\"
ProcedureName[ParameterList] Block.\n");
            out.append("ParameterList ::= \"(\" Parameter {\",\"
Parameter}\")\".\n");
            out.append("Parameter ::= TypeSymbol ParameterTail.\n");
            out.append("ParameterTail ::= VariableName | \"array\" VariableName
\"[\" Constant \"]\".\n");
            out.append("StatementPart ::= {Statement \";\"}.\n");
```

```
            out.append("Statement ::= EmptyStatement | ReadStatement |
WriteStatement | AssignmentStatement | ProcedureStatement | IfStatement |
DoStatement.\n");
            out.append("EmptyStatement ::= \"skip\".\n");
            out.append("ReadStatement ::= \"read\" VariableAccessList.\n");
            out.append("WriteStatement ::= \"write\" ExpressionList.\n");
            out.append("ExpressionList ::= Expression {\",\" Expression}.\n");
            out.append("AssignmentStatement ::= VariableAccessList \":=\"
ExpressionList.\n");
            out.append("ProcedureStatement ::= \"call\" ProcedureName.\n");
            out.append("IfStatement ::= \"if\" GuardedCommandList \"fi\".\n");
            out.append("DoStatement ::= \"do\" GuardedCommandList \"od\".\n");
            out.append("GuardedCommandList ::= GuardedCommand {\"[]\"
GuardedCommand}.\n");
            out.append("GuardedCommand ::= Expression \"->\" StatementPart.\n");
            out.append("Expression ::= PrimaryExpression {PrimaryOperator
PrimaryExpression}.\n");
            out.append("PrimaryOperator ::= \"&\" | \"|\".\n");
            out.append("PrimaryExpression ::= SimpleExpression
[RelationalOperator SimpleExpression].\n");
            out.append("RelationalOperator ::= \"<\" | \"=\" | \">\".\n");
            out.append("SimpleExpression ::= [\"-\"] Term {AddingOperator
Term}.\n");
            out.append("AddingOperator ::= \"+\" | \"-\".\n");
            out.append("Term ::= Factor {MultiplyingOperator Factor}.\n");
            out.append("MultiplyingOperator ::= \"*\" | \"/\" | \"\\\".\n");
            out.append("Factor ::= Constant | VariableAccess | \"(\" Expression
\")\" | \"~\" Factor.\n");
            out.append("VariableAccess ::= VariableName [IndexSelector].\n");
            out.append("VariableAccessList ::= VariableAccess {\",\"
VariableAccess}.\n");
            out.append("IndexSelector ::= \"[\" Expression \"]\".\n");
            out.append("Constant ::= Numeral|BooleanSymbol|ConstantName.\n");
            out.append("Numeral ::= Digit{Digit}.\n");
            out.append("BooleanSymbol ::= \"false\"|\"true\".\n");
            out.append("VariableName ::= Name.\n");
            out.append("ProcedureName ::= Name.");
        return out.toString();
    }

    public String getTokens() {
        StringBuffer out = new StringBuffer();
            out.append("#EMT Lexicon\n");
            out.append("#IGNORED TOKENS\n");
            out.append("SEPERATOR:Seperator\n");
            out.append("\n");
            out.append("#TOKENS\n");
            out.append("\n");
            out.append("DIGIT:Digit\n");
            out.append("LETTER:Letter\n");
            out.append("NAME:Name\n");
            out.append("CONSTANTNAME:ConstantName\n");
            out.append("INTEGERLITERAL:\"integer\"\n");
            out.append("BOOLEANLITERAL:\"boolean\"\n");
            out.append("LPAREN:\"(\"\n");
            out.append("RPAREN:\")\"\n");
            out.append("LBRACKET:\"[\"\n");
```

```
        out.append("RBRACKET:\"]\"\n");
        out.append("BEGIN:\"begin\"\n");
        out.append("END:\"end\"\n");
        out.append("SEMICOLON:\";\"\n");
        out.append("CONST:\"const\"\n");
        out.append("EQUALS:\"=\"\n");
        out.append("ARRAY:\"array\"\n");
        out.append("COMMA:\",\"\n");
        out.append("PERIOD:\".\"\n");
        out.append("PROC:\"proc\"\n");
        out.append("SKIP:\"skip\"\n");
        out.append("READ:\"read\"\n");
        out.append("WRITE:\"write\"\n");
        out.append("BECOMES:\":=\"\n");
        out.append("CALL:\"call\"\n");
        out.append("IFBEGIN:\"if\"\n");
        out.append("IFEND:\"fi\"\n");
        out.append("DOBEGIN:\"do\"\n");
        out.append("DOEND:\"od\"\n");
        out.append("EMPTYARRAY:\"[]\"\n");
        out.append("POINTER:\"->\"\n");
        out.append("AND:\"&\"\n");
        out.append("OR:\"|\"\n");
        out.append("LESSTHAN:\"<\"\n");
        out.append("GREATERTHAN:\">\"\n");
        out.append("MINUS:\"-\"\n");
        out.append("PLUS:\"+\"\n");
        out.append("ASTERIX:\"*\"\n");
        out.append("DIV:\"/\"\n");
        out.append("MOD:\"\\\"\n");
        out.append("NOT:\"~\"\n");
        out.append("FALSEVALUE:\"false\"\n");
        out.append("TRUEVALUE:\"true\"\n");
        out.append("UNDERSCORE:\"_\"\n");
        out.append("\n");
        out.append("#VARIABLES\n");
        out.append("Name:[a-z]([a-zA-Z]|\\d|_)*\n");
        out.append("ConstantName:[A-Z]([A-Z]|\\d|_)*\n");
        out.append("Digit:\\d\n");
        out.append("Letter:[a-zA-Z]\n");
        out.append("Seperator:\\x20|\\t\n");

    return out.toString();
}

public String getSourceCode() {
    StringBuffer out = new StringBuffer();
        out.append("begin\n");
        out.append("  const N = 10;\n");
        out.append("  integer array a[N];\n");
        out.append("  integer x, i;\n");
        out.append("\n");
        out.append("  proc search(integer array a[N])\n");
        out.append("  begin\n");
        out.append("    integer i, m;\n");
        out.append("    boolean found;\n");
        out.append("\n");
```

```
            out.append("    i, m := 0, n;\n");
            out.append("    found := false;\n");
            out.append("    do\n");
            out.append("      i < m ->\n");
            out.append("        if\n");
            out.append("          a[i] = x -> m := i; found := true;[]\n");
            out.append("          ~(a[i] = x) -> i := i + 1;\n");
            out.append("        fi;\n");
            out.append("    od;\n");
            out.append("  end;\n");
            out.append("\n");
            out.append("  i := 0;\n");
            out.append("  do\n");
            out.append("    ~(i > n - 1) -> read a[i]; i := i + 1;\n");
            out.append("  od;\n");
            out.append("  read x;\n");
            out.append("  do\n");
            out.append("    ~(x = 0) ->\n");
            out.append("      call search;\n");
            out.append("      if\n");
            out.append("        found -> write x, i + 1; []\n");
            out.append("        ~found -> write x;\n");
            out.append("      fi;\n");
            out.append("    read x;\n");
            out.append("  od;\n");
            out.append("end.");

        return out.toString();
    }

    public String getASTPackage() {
        return "plast";
    }
    public String getParserPackage() {
        return "plparser";
    }

}
```

defaults.Settings.java

```
/*
 * Created on Sep 15, 2004
 */
package defaults;

/**
 * @author jfoure
 */
public class Settings implements GrammarSettings {
    public String getGrammar() {
        StringBuffer out = new StringBuffer();

        return out.toString();
    }
```

```
    public String getTokens() {
        StringBuffer out = new StringBuffer();

        return out.toString();
    }

    public String getSourceCode() {
        StringBuffer out = new StringBuffer();

        return out.toString();
    }

    public String getASTPackage() {
        return "";
    }
    public String getParserPackage() {
        return "";
    }

}
```

ebnf2ast.ASTAlternativeHelper.java

```
/*
 * Created on Feb 21, 2004
 */
package ebnf2ast;
import java.util.*;
import ebnfast.*;
/**
 * @author jfoure
 */
public class ASTAlternativeHelper {
      List alternativeList = new ArrayList();
      String productionRuleName = null;

      public ASTAlternativeHelper(String name){
            this.productionRuleName = name;
      }

      public void addAlternative(Symbol alternative,boolean canBeEmpty){
            alternativeList.add(new SymbolInfo(alternative,canBeEmpty));
      }

      public String toString(){
            return "ASTAlternativeHelper
["+productionRuleName+","+alternativeList+"]";
      }
}
```

ebnf2ast.ASTSequenceHelper.java

```
/*
 * Created on Feb 21, 2004
 */
package ebnf2ast;
import java.util.*;
import ebnfast.*;
/**
 * @author jfoure
 */
public class ASTSequenceHelper {
      List sequenceList = new ArrayList();
      String productionRuleName = null;

      public ASTSequenceHelper(String name){
            this.productionRuleName = name;
      }

      public void addSequence(Symbol sequence,boolean canBeEmpty){
            sequenceList.add(new SymbolInfo(sequence,canBeEmpty));
      }

      public String toString(){
            return "ASTSequenceHelper
["+productionRuleName+","+sequenceList+"]";
      }
}
```

ebnf2ast.EBNF2ASTConverter.java

```
/*
 * Created on Feb 18, 2004
 */
package ebnf2ast;
import astast.AlternativeDefinition;
import astast.LineDisplay;
import ebnfscanner.*;
import ebnfast.Definition;
import ebnfast.DefinitionBody;
import ebnfast.DefinitionSequence;
import ebnfast.DefinitionTail;
import ebnfast.Display;
import ebnfast.Grammar;
import ebnfast.GroupingDefinition;
import ebnfast.NonTerminal;
import ebnfast.NonTerminalSymbol;
import ebnfast.ProductionRule;
import ebnfast.ProductionRuleList;
import ebnfast.ProductionRuleListTail;
import ebnfast.Symbol;
import ebnfast.SymbolDefinition;
import ebnfast.Terminal;
import ebnfast.TerminalSymbol;
import ebnfast.Visitor;
import ebnfast.ZeroOrMoreDefinition;
import ebnfast.ZeroOrOneDefinition;
```

```
import ebnfparser.*;

import java.util.*;
import common.*;
/**
 * @author jfoure
 */
public class EBNF2ASTConverter implements Visitor{
      TokenI token = null;
      astast.Grammar astGrammar = new astast.Grammar(null,0);
astast.ProductionRuleList currentProductionRuleList = null;
astast.ProductionRuleListTail currentProductionRuleListTail = null;
      //Used to keep track of new production rule names Set
newProductionRuleNames = new HashSet();



      public EBNF2ASTConverter(TokenI token){
            this.token = token;
      }

      /*
Grammar           ::= ProductionRuleList .
ProductionRuleList      ::= ProductionRule ProductionRuleListTail .
ProductionRuleListTail ::= ProductionRule ProductionRuleListTail .
ProductionRule    ::= NonTerminal Definition .
Definition  ::=         NonTerminal AlternativeDefinition SequenceDefinition .
AlternativeDefinition   ::= NonTerminal AlternativeDefinition .
SequenceDefinition      ::= NonTerminal SequenceDefinition .
*/

      public String calculateNewProductionRuleName(String suggestedName){
            String result = suggestedName;
            for (int i=1;newProductionRuleNames.contains(result);i++){
                  result = suggestedName+i;
            }

            newProductionRuleNames.add(result);
            return result;
      }



      //FOR AST RESULT
      private void addProductionRule(astast.ProductionRule productionRule){
            System.out.println("Adding: "+productionRule);
            if (astGrammar.productionRuleList==null){
                  astGrammar.productionRuleList = new
astast.ProductionRuleList(productionRule,null,0);
                  currentProductionRuleList = astGrammar.productionRuleList;
            } else if (currentProductionRuleList.productionRuleListTail==null){
                  currentProductionRuleList.productionRuleListTail = new
astast.ProductionRuleListTail(productionRule,null,0);
                  currentProductionRuleListTail =
currentProductionRuleList.productionRuleListTail;

            }
```

```
            else {
                  currentProductionRuleListTail.productionRuleListTail = new
astast.ProductionRuleListTail(productionRule,null,0);
                  currentProductionRuleListTail =
currentProductionRuleListTail.productionRuleListTail;
            }

      }
      private void addProductionRule(ASTAlternativeHelper alternativeHelper){
            System.out.println("addProductionRule:"+alternativeHelper);
            astast.NonTerminal nonTerminal = new
astast.NonTerminal(alternativeHelper.productionRuleName,0);
            astast.Definition definition = null;
            String variableName;
            SymbolInfo currentSymbolInfo =
(SymbolInfo)alternativeHelper.alternativeList.get(0);
            Symbol currentSymbol = currentSymbolInfo.symbol;
            if (currentSymbol instanceof NonTerminalSymbol){
                  NonTerminalSymbol nonTerminalSymbol = (NonTerminalSymbol)
currentSymbol;
                  variableName = nonTerminalSymbol.nonTerminal.spelling;
                  definition = new astast.Definition(new
astast.NonTerminalSymbol(new
astast.NonTerminal(nonTerminalSymbol.nonTerminal.spelling,0),
nonTerminalSymbol.nonTerminal.spelling, variableName,
currentSymbolInfo.canBeEmpty,0),null,null,0);
            } else
            if (currentSymbol instanceof TerminalSymbol){
                  TerminalSymbol terminalSymbol = (TerminalSymbol)
currentSymbol;
                  variableName =
lookupTokenName(terminalSymbol.terminal.spelling);
                  definition = new astast.Definition(new
astast.TerminalSymbol(new astast.Terminal(terminalSymbol.terminal.spelling,0),
"String", variableName, currentSymbolInfo.canBeEmpty,0),null,null,0);
            }


//          astast.Definition definition = new astast.Definition(new
astast.NonTerminalSymbol(new
astast.NonTerminal(((String)alternativeHelper.alternativeList.get(0)),0),0),null
,null,0);

            AlternativeDefinition currentAlternativeDefinition =
definition.alternativeDefinition;

            if (alternativeHelper.alternativeList.size()>1){
//                definition.alternativeDefinition = new
AlternativeDefinition(new astast.NonTerminalSymbol(new
astast.NonTerminal(((String)alternativeHelper.alternativeList.get(1)),0),0),null
,0);
                  currentSymbolInfo =
(SymbolInfo)alternativeHelper.alternativeList.get(1);
                  currentSymbol = currentSymbolInfo.symbol;
                  if (currentSymbol instanceof NonTerminalSymbol){
                        NonTerminalSymbol nonTerminalSymbol =
(NonTerminalSymbol) currentSymbol;
```

```
                        variableName = nonTerminalSymbol.nonTerminal.spelling;
                        definition.alternativeDefinition = new
AlternativeDefinition(new astast.NonTerminalSymbol(new
astast.NonTerminal(nonTerminalSymbol.nonTerminal.spelling,0),
nonTerminalSymbol.nonTerminal.spelling, variableName,
currentSymbolInfo.canBeEmpty,0),null,0);
                  } else
                  if (currentSymbol instanceof TerminalSymbol){
                        TerminalSymbol terminalSymbol = (TerminalSymbol)
currentSymbol;
                        variableName =
lookupTokenName(terminalSymbol.terminal.spelling);
                        definition.alternativeDefinition = new
AlternativeDefinition(new astast.TerminalSymbol(new
astast.Terminal(terminalSymbol.terminal.spelling,0), "String", variableName,
currentSymbolInfo.canBeEmpty,0),null,0);
                  }
                  currentAlternativeDefinition =
definition.alternativeDefinition;
            }

            for (int i=2;i<alternativeHelper.alternativeList.size();i++){
//                currentAlternativeDefinition.alternativeDefinition = new
AlternativeDefinition(new astast.NonTerminalSymbol(new
astast.NonTerminal(((String)alternativeHelper.alternativeList.get(i)),0),0),null
,0);
                  currentSymbolInfo =
(SymbolInfo)alternativeHelper.alternativeList.get(i);
                  currentSymbol = currentSymbolInfo.symbol;
                  if (currentSymbol instanceof NonTerminalSymbol){
                        NonTerminalSymbol nonTerminalSymbol =
(NonTerminalSymbol) currentSymbol;
                        variableName = nonTerminalSymbol.nonTerminal.spelling;
                        currentAlternativeDefinition.alternativeDefinition = new
AlternativeDefinition(new astast.NonTerminalSymbol(new
astast.NonTerminal(nonTerminalSymbol.nonTerminal.spelling,0),
nonTerminalSymbol.nonTerminal.spelling, variableName,
currentSymbolInfo.canBeEmpty,0),null,0);
                  } else
                  if (currentSymbol instanceof TerminalSymbol){
                        TerminalSymbol terminalSymbol = (TerminalSymbol)
currentSymbol;
                        variableName =
lookupTokenName(terminalSymbol.terminal.spelling);
                        currentAlternativeDefinition.alternativeDefinition = new
AlternativeDefinition(new astast.TerminalSymbol(new
astast.Terminal(terminalSymbol.terminal.spelling,0), "String", variableName,
currentSymbolInfo.canBeEmpty,0),null,0);
                  }
                  currentAlternativeDefinition =
currentAlternativeDefinition.alternativeDefinition;
            }

            astast.ProductionRule productionRule = new
astast.ProductionRule(nonTerminal,definition,0);
            addProductionRule(productionRule);
      }
```

```
/**
 * Take a symbolname and lowercase the first letter and append a number to make
sure it is unique.
 * @param symbolName
 * @param usedVariables
 * @return
 */
private String getVariableName(String symbolName, Set usedVariables){
      String variableName;
      String baseName;

      if (symbolName.length()==1)
            baseName = symbolName.toLowerCase();
      else
            baseName =
symbolName.substring(0,1).toLowerCase()+symbolName.substring(1,symbolName.length
());
      variableName = baseName;
      for (int count=2;usedVariables.contains(variableName);count++){
            variableName = baseName+count;
      }
      usedVariables.add(variableName);
      return variableName;

}

private void addProductionRule(ASTSequenceHelper sequenceHelper){
      System.out.println("addProductionRule:"+sequenceHelper);
      if (sequenceHelper.sequenceList.size()==0)
            return;
      astast.NonTerminal nonTerminal = new
astast.NonTerminal(sequenceHelper.productionRuleName,0);
//    String currentDefinitionBody = (String)sequenceHelper.sequenceList.get(0);
//    astast.Definition definition = null;
//    definition = new astast.Definition(new astast.NonTerminalSymbol(new
astast.NonTerminal(currentDefinitionBody,0),0),null,null,0);
      SymbolInfo currentSymbolInfo =
(SymbolInfo)sequenceHelper.sequenceList.get(0);
      Symbol currentSymbol = currentSymbolInfo.symbol;
      astast.Definition definition = null;

      String variableName;
      Set variableSet = new HashSet();

      if (currentSymbol instanceof NonTerminalSymbol){
            NonTerminalSymbol nonTerminalSymbol = (NonTerminalSymbol)
currentSymbol;
            variableName =
getVariableName(nonTerminalSymbol.nonTerminal.spelling,variableSet);
            definition = new astast.Definition(new astast.NonTerminalSymbol(new
astast.NonTerminal(nonTerminalSymbol.nonTerminal.spelling,0),nonTerminalSymbol.n
onTerminal.spelling, variableName,currentSymbolInfo.canBeEmpty,0),null,null,0);
      } else
      if (currentSymbol instanceof TerminalSymbol){
            TerminalSymbol terminalSymbol = (TerminalSymbol) currentSymbol;
```

```
            variableName =
getVariableName(lookupTokenName(terminalSymbol.terminal.spelling),variableSet)+"
Spelling";
            definition = new astast.Definition(new astast.TerminalSymbol(new
astast.Terminal(terminalSymbol.terminal.spelling,0),"String",
variableName,currentSymbolInfo.canBeEmpty,0),null,null,0);
      }

      astast.SequenceDefinition currentSequenceDefinition = null;
      if (sequenceHelper.sequenceList.size()>1){
//          currentDefinitionBody = (String)sequenceHelper.sequenceList.get(1);
//          definition.sequenceDefinition = new astast.SequenceDefinition(new
astast.NonTerminalSymbol(new
astast.NonTerminal(currentDefinitionBody,0),0),null,0);
            currentSymbolInfo = (SymbolInfo)sequenceHelper.sequenceList.get(1);
            currentSymbol = currentSymbolInfo.symbol;
            if (currentSymbol instanceof NonTerminalSymbol){
                  NonTerminalSymbol nonTerminalSymbol = (NonTerminalSymbol)
currentSymbol;
                  variableName =
getVariableName(nonTerminalSymbol.nonTerminal.spelling,variableSet);
                  definition.sequenceDefinition = new
astast.SequenceDefinition(new astast.NonTerminalSymbol(new
astast.NonTerminal(nonTerminalSymbol.nonTerminal.spelling,0),nonTerminalSymbol.n
onTerminal.spelling, variableName,currentSymbolInfo.canBeEmpty,0),null,0);
            } else
            if (currentSymbol instanceof TerminalSymbol){
                  TerminalSymbol terminalSymbol = (TerminalSymbol)
currentSymbol;
                  variableName =
getVariableName(lookupTokenName(terminalSymbol.terminal.spelling),variableSet)+"
Spelling";
                  definition.sequenceDefinition = new
astast.SequenceDefinition(new astast.TerminalSymbol(new
astast.Terminal(terminalSymbol.terminal.spelling,0),"String",
variableName,currentSymbolInfo.canBeEmpty,0),null,0);
            }

            currentSequenceDefinition = definition.sequenceDefinition;
      }

      for (int i=2;i<sequenceHelper.sequenceList.size();i++){
//          currentDefinitionBody = (String)sequenceHelper.sequenceList.get(i);
//          currentSequenceDefinition.sequenceDefinition = new
astast.SequenceDefinition(new astast.NonTerminalSymbol(new
astast.NonTerminal(currentDefinitionBody,0),0),null,0);
            currentSymbolInfo = (SymbolInfo)sequenceHelper.sequenceList.get(i);
            currentSymbol = currentSymbolInfo.symbol;
            if (currentSymbol instanceof NonTerminalSymbol){
                  NonTerminalSymbol nonTerminalSymbol = (NonTerminalSymbol)
currentSymbol;
                  variableName =
getVariableName(nonTerminalSymbol.nonTerminal.spelling,variableSet);
                  currentSequenceDefinition.sequenceDefinition = new
astast.SequenceDefinition(new astast.NonTerminalSymbol(new
astast.NonTerminal(nonTerminalSymbol.nonTerminal.spelling,0),nonTerminalSymbol.n
onTerminal.spelling, variableName,currentSymbolInfo.canBeEmpty,0),null,0);
```

```
            } else
            if (currentSymbol instanceof TerminalSymbol){
                  TerminalSymbol terminalSymbol = (TerminalSymbol)
currentSymbol;
                  variableName =
getVariableName(lookupTokenName(terminalSymbol.terminal.spelling),variableSet)+"
Spelling";
                  currentSequenceDefinition.sequenceDefinition = new
astast.SequenceDefinition(new astast.TerminalSymbol(new
astast.Terminal(terminalSymbol.terminal.spelling,0),"String",
variableName,currentSymbolInfo.canBeEmpty,0),null,0);
            }

            currentSequenceDefinition =
currentSequenceDefinition.sequenceDefinition;
      }

      astast.ProductionRule productionRule = new
astast.ProductionRule(nonTerminal,definition,0);
      addProductionRule(productionRule);
}


      private void addProductionRule(ProductionRule productionRule){

      addProductionRule(productionRule.nonTerminal.spelling,productionRule.defin
ition);
      }


      private boolean isSingleDefinition(Definition definition){
            if (definition.definitionSequence.definitionBody instanceof
SymbolDefinition && definition.definitionSequence.definitionSequence==null)
                  return true;
            return false;
      }

      /*
       * Get the Token name of special characters
       */
      private String lookupTokenName(String spelling){
//          if (token.lookupConstantSpelling(spelling)!=null)
//                return token.lookupConstantSpelling(spelling);
            if (token.lookupVariableSpelling(spelling)!=null)
                  return token.lookupVariableSpelling(spelling);
            else
                  return spelling;
      }


      /**
       * Get the name of the first Symbol in a definition.
       *
       */
      private String getSingleDefinitionName(DefinitionSequence
definitionSequence){
            if (definitionSequence.definitionBody instanceof SymbolDefinition){
```

```
                  SymbolDefinition symbolDefinition = (SymbolDefinition)
definitionSequence.definitionBody;
                  if (symbolDefinition.symbol instanceof NonTerminalSymbol){
                        NonTerminalSymbol nonTerminalSymbol =
(NonTerminalSymbol) symbolDefinition.symbol;
                        return
lookupTokenName(nonTerminalSymbol.nonTerminal.spelling);
                  }
                  else if (symbolDefinition.symbol instanceof TerminalSymbol){
                        TerminalSymbol terminalSymbol = (TerminalSymbol)
symbolDefinition.symbol;
                        return
lookupTokenName(terminalSymbol.terminal.spelling);
                  }
            }
            else if (definitionSequence.definitionBody instanceof
ZeroOrMoreDefinition){
                  return
getSingleDefinitionName(((ZeroOrMoreDefinition)definitionSequence.definitionBody
).definition.definitionSequence);
            }
            else if (definitionSequence.definitionBody instanceof
ZeroOrOneDefinition){
                  return
getSingleDefinitionName(((ZeroOrOneDefinition)definitionSequence.definitionBody)
.definition.definitionSequence);
            }
            else if (definitionSequence.definitionBody instanceof
GroupingDefinition){
                  return
getSingleDefinitionName(((GroupingDefinition)definitionSequence.definitionBody).
definition.definitionSequence);
            }


            return null;
      }



      //check for alternatives
      private void addProductionRule(String name, Definition definition){
            addProductionRule(name,definition,false);
      }

      /**
       * Return true if definition only contains single terminal alternatives.
       * @param definition
       * @return
       */
      public static boolean isBasicAlternative(Definition definition){
            //Definition currentDefinition = definition;
            DefinitionTail currentDefinitionTail = definition.definitionTail;
            DefinitionSequence currentDefinitionSequence =
definition.definitionSequence;
            do{
```

```
                  if (!(currentDefinitionSequence.definitionBody instanceof
SymbolDefinition))
                        return false;
                  if
      (!(((SymbolDefinition)currentDefinitionSequence.definitionBody).symbol
instanceof TerminalSymbol) )
                        return false;
                  if (currentDefinitionSequence.definitionSequence!=null)
                        return false;

                  if (currentDefinitionTail!=null){
                        currentDefinitionSequence =
currentDefinitionTail.definitionSequence;
                        currentDefinitionTail =
currentDefinitionTail.definitionTail;
                  }
                  else
                        currentDefinitionSequence = null;
            } while (currentDefinitionSequence!=null);
            return true;
      }

      private void addProductionRule(String name, Definition definition, boolean
isTail){
            System.out.println("Adding: "+name+"::="+definition);
            if (definition.definitionTail!=null){
                  System.out.println("Contains alternatives:"+name);
                  ASTAlternativeHelper alternativeHelper = new
ASTAlternativeHelper(name);
                  if (isBasicAlternative(definition)){
                        System.out.println("Basic alternatives!!!");
                        DefinitionTail currentDefinitionTail =
definition.definitionTail;
                        DefinitionSequence currentDefinitionSequence =
definition.definitionSequence;
                        do{
                              TerminalSymbol terminalSymbol =
(TerminalSymbol)(((SymbolDefinition)currentDefinitionSequence.definitionBody).sy
mbol);

      alternativeHelper.addAlternative(terminalSymbol,false);
                              if (currentDefinitionTail!=null){
                                    currentDefinitionSequence =
currentDefinitionTail.definitionSequence;
                                    currentDefinitionTail =
currentDefinitionTail.definitionTail;
                              }
                              else
                                    currentDefinitionSequence = null;
                        } while (currentDefinitionSequence!=null);
addProductionRule(alternativeHelper);
                  } else{
                        List sequenceList = new ArrayList();
                        String definitionName =
name+getSingleDefinitionName(definition.definitionSequence);
                        alternativeHelper.addAlternative(new
NonTerminalSymbol(new NonTerminal(definitionName,0),0),false);
```

```
                        System.out.println("definitionName1:"+definitionName);
                        sequenceList.add(new
SequenceDefinitionHelper(definitionName,definition.definitionSequence));
                        DefinitionTail currentDefinitionTail =
definition.definitionTail;
                        int count = 2;
                        while (currentDefinitionTail!=null){
                              definitionName =
name+getSingleDefinitionName(currentDefinitionTail.definitionSequence);

      System.out.println("definitionName2:"+definitionName);
                              alternativeHelper.addAlternative(new
NonTerminalSymbol(new NonTerminal(definitionName,0),0),false);
                              sequenceList.add(new
SequenceDefinitionHelper(definitionName,currentDefinitionTail.definitionSequence
));
                              count++;
                              currentDefinitionTail =
currentDefinitionTail.definitionTail;
                        }

                        if (isTail){
//                            alternativeHelper.addAlternative(name);

                              alternativeHelper.addAlternative(new
NonTerminalSymbol(new NonTerminal(name,0),0),false);
                        }

                        addProductionRule(alternativeHelper);
                        for (int i=0;i<sequenceList.size();i++){
                              SequenceDefinitionHelper helper =
(SequenceDefinitionHelper) sequenceList.get(i);

      addProductionRule(helper.definitionName,helper.definitionSequence);
                        }
                  }
            } else {
                  addProductionRule(name,definition.definitionSequence,isTail);
            }

      }

      //just for sequences
      private void addProductionRule(String name, DefinitionSequence
definitionSequence){
            addProductionRule(name,definitionSequence,false);
      }

      private void addProductionRule(String name, DefinitionSequence
definitionSequence,boolean isTail){
            System.out.println("Contains NOT alternatives");
            ASTSequenceHelper sequenceHelper = new ASTSequenceHelper(name);
            List definitionList = new ArrayList();
            DefinitionBody currentDefinitionBody = null;
            //String currentSpelling = null;
            DefinitionSequence currentDefinitionSequence = definitionSequence;
```

```
            do{
                  currentDefinitionBody =
currentDefinitionSequence.definitionBody;
                  if (currentDefinitionBody instanceof ZeroOrMoreDefinition){
                        ZeroOrMoreDefinition zeroOrMoreDefinition =
(ZeroOrMoreDefinition) currentDefinitionBody;

      //addProductionRule(sequenceHelper.productionRuleName+"ZeroOrMore",zeroOrM
oreDefinition.definition,true);
//                        String nonTerminalName =
sequenceHelper.productionRuleName+getSingleDefinitionName(currentDefinitionSeque
nce)+"ZeroOrMore";
                        String nonTerminalName =
calculateNewProductionRuleName(sequenceHelper.productionRuleName+getSingleDefini
tionName(currentDefinitionSequence)+"ZeroOrMore");
                        definitionList.add(new
DefinitionHelper(nonTerminalName,zeroOrMoreDefinition.definition,true));

      System.out.println("!!!ZeroOrMoreDefinition:"+currentDefinitionBody);
                        sequenceHelper.addSequence(new NonTerminalSymbol(new
NonTerminal(nonTerminalName,0),0),true);
                  }
                  else if (currentDefinitionBody instanceof
ZeroOrOneDefinition){
                        ZeroOrOneDefinition zeroOrOneDefinition =
(ZeroOrOneDefinition) currentDefinitionBody;

      //addProductionRule(sequenceHelper.productionRuleName+"ZeroOrOne",zeroOrOn
eDefinition.definition);
//                        String nonTerminalName =
sequenceHelper.productionRuleName+getSingleDefinitionName(currentDefinitionSeque
nce)+"ZeroOrOne";
                        String nonTerminalName =
calculateNewProductionRuleName(sequenceHelper.productionRuleName+getSingleDefini
tionName(currentDefinitionSequence)+"ZeroOrOne");
                        definitionList.add(new
DefinitionHelper(nonTerminalName,zeroOrOneDefinition.definition,false));

      System.out.println("!!!ZeroOrOneDefinition:"+currentDefinitionBody);
                        sequenceHelper.addSequence(new NonTerminalSymbol(new
NonTerminal(nonTerminalName,0),0),true);
                  }
                  else if (currentDefinitionBody instanceof GroupingDefinition){
                        GroupingDefinition groupingDefinition =
(GroupingDefinition) currentDefinitionBody;
//
      addProductionRule(sequenceHelper.productionRuleName+"Group",groupingDefini
tion.definition);

//                        String nonTerminalName =
sequenceHelper.productionRuleName+getSingleDefinitionName(currentDefinitionSeque
nce)+"Group";
                        String nonTerminalName =
calculateNewProductionRuleName(sequenceHelper.productionRuleName+getSingleDefini
tionName(currentDefinitionSequence)+"Group");
                        definitionList.add(new
DefinitionHelper(nonTerminalName,groupingDefinition.definition,false));
```

```
      System.out.println("!!!GroupingDefinition:"+currentDefinitionBody);
// sequenceHelper.addSequence(sequenceHelper.productionRuleName+"Group");
                        sequenceHelper.addSequence(new NonTerminalSymbol(new
NonTerminal(nonTerminalName,0),0),false);
                  }
                  else if (currentDefinitionBody instanceof SymbolDefinition){
                        SymbolDefinition symbolDefinition = (SymbolDefinition)
currentDefinitionBody;
                        if (symbolDefinition.symbol instanceof
NonTerminalSymbol){
                              NonTerminalSymbol nonTerminalSymbol =
(NonTerminalSymbol) symbolDefinition.symbol;
//
      sequenceHelper.addSequence(nonTerminalSymbol.nonTerminal.spelling);

      sequenceHelper.addSequence(nonTerminalSymbol,false);
                        } else {
                              TerminalSymbol terminalSymbol = (TerminalSymbol)
symbolDefinition.symbol;
//
      sequenceHelper.addSequence(terminalSymbol.terminal.spelling);
                              sequenceHelper.addSequence(terminalSymbol,false);
                        }
                  }


            currentDefinitionSequence =
currentDefinitionSequence.definitionSequence;
            } while (currentDefinitionSequence!=null);

            if (isTail){
                  sequenceHelper.addSequence(new NonTerminalSymbol(new
NonTerminal(name,0),0),true);
            }

            addProductionRule(sequenceHelper);
            for (int i=0;i<definitionList.size();i++){
                  DefinitionHelper helper = (DefinitionHelper)
definitionList.get(i);

      addProductionRule(helper.definitionName,helper.definition,helper.canBeEmpt
y);
            }
      }




      public void check(Grammar grammar){
            grammar.visit(this, null);
      }
      public Object visitGrammar(Grammar grammar, Object arg, int line){
            grammar.productionRuleList.visit(this,null); return null;
      }
```

```
      public Object visitProductionRuleList(ProductionRuleList
productionrulelist, Object arg, int line){
            productionrulelist.productionRule.visit(this,null); if
(productionrulelist.productionRuleListTail!=null)
                  productionrulelist.productionRuleListTail.visit(this,null);
            return null;
      }
      public Object visitProductionRuleListTail(ProductionRuleListTail
productionrulelisttail, Object arg, int line){
            productionrulelisttail.productionRuleList.visit(this,null); if
(productionrulelisttail.productionRuleListTail!=null)

      productionrulelisttail.productionRuleListTail.visit(this,null);
            return null;
      }
      public Object visitProductionRule(ProductionRule productionrule, Object
arg, int line){
            System.out.println("ProductionRule:"+productionrule);
            addProductionRule(productionrule);
            productionrule.nonTerminal.visit(this,null);
//          productionrule.definition.visit(this,productionrule.nonTerminal);
            productionrule.definition.visit(this,null);
            return null;
      }
      public Object visitDefinition(Definition definition, Object arg, int
line){
//          NonTerminal nonTerminal = (NonTerminal) arg;
//          System.out.println("Definition:"+nonTerminal+"::="+definition);

            definition.definitionSequence.visit(this,null);
            if (definition.definitionTail!=null)
                  definition.definitionTail.visit(this,null);
            return null;
      }
      public Object visitDefinitionTail(DefinitionTail definitiontail, Object
arg, int line){
            definitiontail.definitionSequence.visit(this,null); if
(definitiontail.definitionTail!=null)
                  definitiontail.definitionTail.visit(this,null);
            return null;
      }
      public Object visitDefinitionSequence(DefinitionSequence
definitionsequence, Object arg, int line){
            definitionsequence.definitionBody.visit(this,null); if
(definitionsequence.definitionSequence!=null)
                  definitionsequence.definitionSequence.visit(this,null);
            return null;
      }
      public Object visitDefinitionBody(DefinitionBody definitionbody, Object
arg, int line){
            return null;
      }
      public Object visitSymbolDefinition(SymbolDefinition symboldefinition,
Object arg, int line){
            symboldefinition.symbol.visit(this,null); return null;
      }
```

```
	public Object visitGroupingDefinition(GroupingDefinition
groupingdefinition, Object arg, int line){
		groupingdefinition.definition.visit(this,null); return null;
	}
	public Object visitZeroOrOneDefinition(ZeroOrOneDefinition
zerooronedefinition, Object arg, int line){
		zerooronedefinition.definition.visit(this,null); return null;
	}
	public Object visitZeroOrMoreDefinition(ZeroOrMoreDefinition
zeroormoredefinition, Object arg, int line){
		zeroormoredefinition.definition.visit(this,null); return null;
	}
	public Object visitSymbol(Symbol symbol, Object arg, int line){
		return null;
	}
	public Object visitTerminalSymbol(TerminalSymbol terminalsymbol, Object
arg, int line){
		terminalsymbol.terminal.visit(this,null); return null;
	}
	public Object visitNonTerminalSymbol(NonTerminalSymbol nonterminalsymbol,
Object arg, int line){
		nonterminalsymbol.nonTerminal.visit(this,null); return null;
	}
	public Object visitNonTerminal(NonTerminal nonterminal, Object arg, int
line){
		return null;
	}
	public Object visitTerminal(Terminal terminal, Object arg, int line){
		return null;
	}

	public static void main(String[] args) throws Exception{
		Parser parser = new Parser();
//		Grammar grammar = parser.parse("A::=B. B::=C D | \"e\". C::=B.");
		Grammar grammar = parser.parse("A::= \"b\" { \"c\" } | B. B::= ( C D
| E) F. C::= [ \"d\" ] \"e\". D::= { \"D\" }. E::= \"f\" \"g\". F::= [ ( \"h\"
\"i\" ) ]. G::= \"a\" | \"b\" | \"c\" | \"d\" | \"e\" | \"f\".");
//		Grammar grammar = parser.parse();
		new Display(grammar);
		System.out.println("Starting EBNF2ASTConverter");
		EBNF2ASTConverter ebnf2ASTConverter = new EBNF2ASTConverter(new
emt.Token());
		ebnf2ASTConverter.check(grammar);
		System.out.println("Displaying line:");
		//new astast.Display(ebnf2ASTConverter.astGrammar);
		System.out.println("********");
		new astast.LineDisplay(ebnf2ASTConverter.astGrammar);
		System.out.println("The program is terminated normally.");

	}
	/**
	 * @return
	 */
	public astast.Grammar getAstGrammar() {
		return astGrammar;
	}
```

```
	/**
	 * @param grammar
	 */
	public void setAstGrammar(astast.Grammar grammar) {
		astGrammar = grammar;
	}


	class SequenceDefinitionHelper{
		String definitionName;
		DefinitionSequence definitionSequence;
		SequenceDefinitionHelper(String definitionName, DefinitionSequence
definitionSequence){
			this.definitionName = definitionName;
			this.definitionSequence = definitionSequence;
		}
	}
	class DefinitionHelper{
		String definitionName;
		Definition definition;
		boolean canBeEmpty;

		DefinitionHelper(String definitionName, Definition definition,
boolean canBeEmpty){
			this.definitionName = definitionName;
			this.definition = definition;
			this.canBeEmpty = canBeEmpty;
		}
	}

}
```

ebnf2ast.SymbolInfo.java

```
/*
 * Created on Mar 21, 2004
 */
package ebnf2ast;
import ebnfast.*;
/**
 * @author jfoure
 */
public class SymbolInfo {
	Symbol symbol;
	boolean canBeEmpty;

	public SymbolInfo(Symbol symbol, boolean canBeEmpty){
		this.symbol = symbol;
		this.canBeEmpty = canBeEmpty;
	}

}
```

ebnf2parser.EBNF2Parser.java

```
package ebnf2parser;
import ebnfscanner.*;
import ebnfparser.*;
import ebnfast.*;
import ebnfsets.*;
import common.*;

import java.util.*;
import java.io.*;

public class EBNF2Parser implements Visitor{
      StarterSetTable starterSetTable = null;
      Set undefinedNonTerminalSet = null;
      TokenI token = null;
      StringBuffer out = new StringBuffer();

      public EBNF2Parser(StarterSetTable starterSetTable,Set
undefinedNonTerminalSet, TokenI token){
            this.starterSetTable = starterSetTable;
            this.token = token;
            this.undefinedNonTerminalSet = undefinedNonTerminalSet;
      }


      public void check(Grammar grammar){
            grammar.visit(this, null);
      }
      public Object visitGrammar(Grammar grammar, Object arg, int line){
            grammar.productionRuleList.visit(this,null); return null;
      }
      public Object visitProductionRuleList(ProductionRuleList
productionrulelist, Object arg, int line){
            productionrulelist.productionRule.visit(this,null); if
(productionrulelist.productionRuleListTail!=null)
                  productionrulelist.productionRuleListTail.visit(this,null);
            return null;
      }
      public Object visitProductionRuleListTail(ProductionRuleListTail
productionrulelisttail, Object arg, int line){
            productionrulelisttail.productionRuleList.visit(this,null); if
(productionrulelisttail.productionRuleListTail!=null)

      productionrulelisttail.productionRuleListTail.visit(this,null);
            return null;
      }
      public Object visitProductionRule(ProductionRule productionrule, Object
arg, int line){
            out.append("//ProductionRule:"+productionrule+"\n");
            out.append("private void
parse"+productionrule.nonTerminal.spelling+"(){\n");
            productionrule.nonTerminal.visit(this,null);
productionrule.definition.visit(this,null);
            out.append("}\n");

            return null;
```

```
	}
	public Object visitDefinition(Definition definition, Object arg, int
line){
		if (definition.definitionTail==null){
			DefinitionSequenceHelper definitionSequenceHelper = new
DefinitionSequenceHelper();
			definitionSequenceHelper.setAlternative(false);

	definition.definitionSequence.visit(this,definitionSequenceHelper);
		}
		else if (definition.definitionTail!=null){
			DefinitionSequenceHelper definitionSequenceHelper = new
DefinitionSequenceHelper();
			definitionSequenceHelper.setAlternative(true);
			out.append("\tswitch (currentToken.kind){\n");

	definition.definitionSequence.visit(this,definitionSequenceHelper);
			out.append("\t\tbreak;\n");
			definition.definitionTail.visit(this,null);
			out.append("\t\tdefault:\n\t\tnew Error(\"Syntax error.\",
currentToken.line);\n\t}\n");
		}

		return null;
	}

	public Object visitDefinitionTail(DefinitionTail definitiontail, Object
arg, int line){
		//System.out.println("**Alternative**");
		DefinitionSequenceHelper definitionSequenceHelper = new
DefinitionSequenceHelper();
		definitionSequenceHelper.setAlternative(true);

	definitiontail.definitionSequence.visit(this,definitionSequenceHelper);
		out.append("\t\tbreak;\n");
		if (definitiontail.definitionTail!=null)
			definitiontail.definitionTail.visit(this,null);
		return null;
	}

	public Object visitDefinitionSequence(DefinitionSequence
definitionsequence, Object arg, int line){
		DefinitionSequenceHelper definitionSequenceHelper = (arg==null?new
DefinitionSequenceHelper():((DefinitionSequenceHelper)arg));
//		System.out.println("**Start Sequence");
System.out.println("definitionSequenceHelper:"+definitionSequenceHelper.isAltern
ative()+":"+definitionSequenceHelper.getCount());
		//only display the case if it is the first one in the sequence
		if (definitionSequenceHelper.isAlternative() &&
(definitionSequenceHelper.getCount()==0 ||
definitionSequenceHelper.isStartsWithEmpty()) ){
//			StarterSet starterSet =
StarterSetChecker.findStarterSet(definitionsequence,starterSetTable);
			StarterSet starterSet =
StarterSetChecker.findStarterSet(definitionsequence.definitionBody,starterSetTab
le);
			System.out.println("starterSet:"+starterSet);
```

```
                  if (!starterSet.isCanBeEmpty()){
                        definitionSequenceHelper.setStartsWithEmpty(false);
                  }

                  Iterator starterSetIterator = starterSet.symbolSet.iterator();
                  while (starterSetIterator.hasNext()){
                        Symbol starterSetSymbol =
(Symbol)starterSetIterator.next();
                        if (starterSetSymbol instanceof NonTerminalSymbol){
                              NonTerminalSymbol nonTerminalSymbol =
(NonTerminalSymbol)starterSetSymbol;
                              out.append("\t\tcase
Token."+token.lookupConstantSpelling(nonTerminalSymbol.nonTerminal.spelling)+":\
n");
                        } else {
                              TerminalSymbol terminalSymbol =
(TerminalSymbol)starterSetSymbol;
                              out.append("\t\tcase
Token."+token.lookupConstantSpelling(terminalSymbol.terminal.spelling)+":\n");
                        }
                  }

            }

            definitionsequence.definitionBody.visit(this,null);
//          System.out.println("**End Sequence");
            if (definitionsequence.definitionSequence!=null){
                  definitionSequenceHelper.incCount();

      definitionsequence.definitionSequence.visit(this,definitionSequenceHelper)
;
            }


            return null;
      }
      //never called
      public Object visitDefinitionBody(DefinitionBody definitionbody, Object
arg, int line){
            //System.out.println("visitDefinitionBody:"+definitionbody);
            return null;
      }
      public Object visitSymbolDefinition(SymbolDefinition symboldefinition,
Object arg, int line){
            //System.out.println("visitSymbolDefinition:"+symboldefinition);
            if (symboldefinition.symbol instanceof NonTerminalSymbol){
                  NonTerminalSymbol nonTerminalSymbol =
(NonTerminalSymbol)symboldefinition.symbol;

      out.append("\t\t\tparse"+nonTerminalSymbol.nonTerminal.spelling+"();\n");
            } else {
                  TerminalSymbol terminalSymbol =
(TerminalSymbol)symboldefinition.symbol;

      out.append("\t\t\taccept(Token."+token.lookupConstantSpelling(terminalSymb
ol.terminal.spelling)+");\n");
            }
```

```
            symboldefinition.symbol.visit(this,null);
            return null;
      }
      public Object visitGroupingDefinition(GroupingDefinition
groupingdefinition, Object arg, int line){
            groupingdefinition.definition.visit(this,null); return null;
      }
      public Object visitZeroOrOneDefinition(ZeroOrOneDefinition
zerooronedefinition, Object arg, int line){
            StarterSet starterSet =
StarterSetChecker.findStarterSet(zerooronedefinition.definition.definitionSequen
ce,starterSetTable);
            out.append("\t\t\tif (");
            Iterator starterSetIterator = starterSet.symbolSet.iterator();
            while (starterSetIterator.hasNext()){
                  Symbol starterSetSymbol = (Symbol)starterSetIterator.next();
                  if (starterSetSymbol instanceof NonTerminalSymbol){
                        NonTerminalSymbol nonTerminalSymbol =
(NonTerminalSymbol)starterSetSymbol;

      out.append("currentToken.kind==Token."+token.lookupConstantSpelling(nonTer
minalSymbol.nonTerminal.spelling));
                  } else {
                        TerminalSymbol terminalSymbol =
(TerminalSymbol)starterSetSymbol;

      out.append("currentToken.kind==Token."+token.lookupConstantSpelling(termin
alSymbol.terminal.spelling));
                  }

                  if (starterSetIterator.hasNext())
                        out.append(" || ");
            }
            out.append("){\n");
            zerooronedefinition.definition.visit(this,null);
            out.append("\t\t\t}\n");
            return null;
      }
      public Object visitZeroOrMoreDefinition(ZeroOrMoreDefinition
zeroormoredefinition, Object arg, int line){
            StarterSet starterSet =
StarterSetChecker.findStarterSet(zeroormoredefinition.definition.definitionSeque
nce,starterSetTable);
            out.append("\t\t\twhile (");
            Iterator starterSetIterator = starterSet.symbolSet.iterator();
            while (starterSetIterator.hasNext()){
                  Symbol starterSetSymbol = (Symbol)starterSetIterator.next();
                  if (starterSetSymbol instanceof NonTerminalSymbol){
                        NonTerminalSymbol nonTerminalSymbol =
(NonTerminalSymbol)starterSetSymbol;

      out.append("currentToken.kind==Token."+token.lookupConstantSpelling(nonTer
minalSymbol.nonTerminal.spelling));
                  } else {
                        TerminalSymbol terminalSymbol =
(TerminalSymbol)starterSetSymbol;
```

```
      out.append("currentToken.kind==Token."+token.lookupConstantSpelling(termin
alSymbol.terminal.spelling));
                  }

                  if (starterSetIterator.hasNext())
                        out.append(" || ");
            }
            out.append("){\n");
            zeroormoredefinition.definition.visit(this,null);
            out.append("\t\t\t}\n");
            return null;
      }
      public Object visitSymbol(Symbol symbol, Object arg, int line){
            return null;
      }
      public Object visitTerminalSymbol(TerminalSymbol terminalsymbol, Object
arg, int line){
            terminalsymbol.terminal.visit(this,null); return null;
      }
      public Object visitNonTerminalSymbol(NonTerminalSymbol nonterminalsymbol,
Object arg, int line){
            nonterminalsymbol.nonTerminal.visit(this,null); return null;
      }
      public Object visitNonTerminal(NonTerminal nonterminal, Object arg, int
line){
            return null;
      }
      public Object visitTerminal(Terminal terminal, Object arg, int line){
            return null;
      }

      public static void main(String[] args) throws Exception{
            Parser parser = new Parser();
            Grammar grammar = parser.parse();
            //new Display(grammar);

            StarterSetChecker starterSetChecker = new StarterSetChecker();
            starterSetChecker.check(grammar);
            System.out.println("*****");
            StarterSetTable starterSetTable =
starterSetChecker.findStarterSet();

            System.out.println("*****");
            NonTerminalChecker nonTerminalChecker = new NonTerminalChecker();
            nonTerminalChecker.check(grammar);

      System.out.println("definedNonTerminalSet:"+nonTerminalChecker.getDefinedN
onTerminalSet());

      System.out.println("nonTerminalSet:"+nonTerminalChecker.getUndefinedNonTer
minalSet());

            System.out.println("Generate the parser");

//emt
```

```
            EBNF2Parser ebnf2Parser = new
EBNF2Parser(starterSetTable,nonTerminalChecker.getUndefinedNonTerminalSet(),new
emt.Token());
//minitriangle
//          EBNF2Parser ebnf2Parser = new
EBNF2Parser(starterSetTable,nonTerminalChecker.getUndefinedNonTerminalSet(),new
minitriangle.Token());
            ebnf2Parser.check(grammar);

//emt
            ebnf2Parser.generateParser("C:\\Documents and
Settings\\jfoure\\Desktop\\cs605\\Thesis\\ebnf\\emtgenerated\\Parser.java","emtg
enerated");
//minitriangle
//          ebnf2Parser.generateParser("C:\\Documents and
Settings\\jfoure\\Desktop\\cs605\\Thesis\\ebnf\\minitrianglegenerated\\Parser.ja
va","minitrianglegenerated");

            System.out.println("The program is terminated normally.");

      }


      private void generateParser(String filename,String packageName){
            StringBuffer header = new StringBuffer();
            header.append("package "+packageName+";\n\n"+
                  "public class Parser{\n"+
                  "\tprivate Token currentToken;\n"+
                  "\tScanner scanner;\n\n"+
                  "\tprivate void accept(byte expectedKind){\n"+
            "\t\tif(currentToken.kind == expectedKind)\n"+
                  "\t\t\tcurrentToken = scanner.scan();\n"+
            "\t\telse\n"+
                  "\t\tnew Error(\"Syntax error: \" + currentToken.spelling + \"
is not expected.\",currentToken.line);\n"+
                  "\t}\n\n"+
                  "\tprivate void acceptIt(){\n"+
            "\t\tcurrentToken = scanner.scan();\n"+
                  "\t}\n\n"+
                  "\tpublic void parse(){\n"+
            "\t\tSourceFile sourceFile = new SourceFile();\n"+
            "\t\tscanner = new Scanner(sourceFile.openFile());\n"+
            "\t\tcurrentToken = scanner.scan();\n"+
            "\t\tparseProgram();\n"+
            "\t\tif(currentToken.kind != Token.EOT)\n"+
                  "\t\t\tnew Error(\"Syntax error: Redundant characters at the
end of program.\",currentToken.line);\n"+
                  "\t\t}\n\n");
                  Iterator undefinedNonTerminalSetIterator =
undefinedNonTerminalSet.iterator();
                  while (undefinedNonTerminalSetIterator.hasNext()){
                        String undefinedNonTerminal = (String)
undefinedNonTerminalSetIterator.next();
                        header.append(
                              "\tpublic void
parse"+undefinedNonTerminal+"(){\n"+
```

```
      "\t\taccept(Token."+token.lookupConstantSpelling(undefinedNonTerminal)+");
\n"+
                                    "\t}\n\n"
                              );
                        }

                  out.insert(0,header);
                  out.append("\n}");


                  writeOutput(filename,out.toString());
            }

            private static void writeOutput(String filename, String data){
                  try{
//                      FileWriter out = new FileWriter("C:\\Documents and
Settings\\jfoure\\Desktop\\cs605\\Thesis\\ebnf\\astast\\"+filename+".java");
                        FileWriter out = new FileWriter(filename);
                        out.write(data);
                        out.flush();
                        out.close();
                  } catch (Exception e) {
                        e.printStackTrace();
                  }

            }


            class DefinitionSequenceHelper{
                  int count = 0;
                  boolean isAlternative = false;
                  boolean startsWithEmpty = true;

                  /**
                   * @return
                   */
                  public int getCount() {
                        return count;
                  }

                  /**
                   * @return
                   */
                  public boolean isAlternative() {
                        return isAlternative;
                  }

                  /**
                   * @param i
                   */
                  public void setCount(int i) {
                        count = i;
                  }

                  /**
                   * @param b
```

```
             */
            public void setAlternative(boolean b) {
                  isAlternative = b;
            }

            public void incCount(){
                  ++count;
            }
            /**
             * @return
             */
            public boolean isStartsWithEmpty() {
                  return startsWithEmpty;
            }

            /**
             * @param b
             */
            public void setStartsWithEmpty(boolean b) {
                  startsWithEmpty = b;
            }

      }
}
```

ebnf2parser.NonTerminalChecker.java

```
package ebnf2parser;
import ebnfast.*;
import ebnfscanner.*;
import ebnfparser.*;

import java.util.*;
public class NonTerminalChecker implements Visitor{
      Set definedNonTerminalSet = new HashSet();
      Set nonTerminalSet = new HashSet();
      public Set getDefinedNonTerminalSet(){
            return definedNonTerminalSet;
      }
      public Set getUndefinedNonTerminalSet(){
            return nonTerminalSet;
      }

      public void check(Grammar grammar){
            grammar.visit(this, null);
            nonTerminalSet.removeAll(definedNonTerminalSet);

      }
      public Object visitGrammar(Grammar grammar, Object arg, int line){
            grammar.productionRuleList.visit(this,null); return null;
      }
      public Object visitProductionRuleList(ProductionRuleList
productionrulelist, Object arg, int line){
```

```
            productionrulelist.productionRule.visit(this,null); if
(productionrulelist.productionRuleListTail!=null)
                  productionrulelist.productionRuleListTail.visit(this,null);
            return null;
      }
      public Object visitProductionRuleListTail(ProductionRuleListTail
productionrulelisttail, Object arg, int line){
            productionrulelisttail.productionRuleList.visit(this,null); if
(productionrulelisttail.productionRuleListTail!=null)

      productionrulelisttail.productionRuleListTail.visit(this,null);
            return null;
      }
      public Object visitProductionRule(ProductionRule productionrule, Object
arg, int line){
            definedNonTerminalSet.add(productionrule.nonTerminal.spelling);
            productionrule.nonTerminal.visit(this,null);
productionrule.definition.visit(this,null); return null;
      }
      public Object visitDefinition(Definition definition, Object arg, int
line){
            definition.definitionSequence.visit(this,null); if
(definition.definitionTail!=null)
                  definition.definitionTail.visit(this,null);
            return null;
      }
      public Object visitDefinitionTail(DefinitionTail definitiontail, Object
arg, int line){
            definitiontail.definitionSequence.visit(this,null); if
(definitiontail.definitionTail!=null)
                  definitiontail.definitionTail.visit(this,null);
            return null;
      }
      public Object visitDefinitionSequence(DefinitionSequence
definitionsequence, Object arg, int line){
            definitionsequence.definitionBody.visit(this,null); if
(definitionsequence.definitionSequence!=null)
                  definitionsequence.definitionSequence.visit(this,null);
            return null;
      }
      public Object visitDefinitionBody(DefinitionBody definitionbody, Object
arg, int line){
            return null;
      }
      public Object visitSymbolDefinition(SymbolDefinition symboldefinition,
Object arg, int line){
            symboldefinition.symbol.visit(this,null); return null;
      }
      public Object visitGroupingDefinition(GroupingDefinition
groupingdefinition, Object arg, int line){
            groupingdefinition.definition.visit(this,null); return null;
      }
      public Object visitZeroOrOneDefinition(ZeroOrOneDefinition
zerooronedefinition, Object arg, int line){
            zerooronedefinition.definition.visit(this,null); return null;
      }
```

```
	public Object visitZeroOrMoreDefinition(ZeroOrMoreDefinition
zeroormoredefinition, Object arg, int line){
		zeroormoredefinition.definition.visit(this,null); return null;
	}
	public Object visitSymbol(Symbol symbol, Object arg, int line){
		return null;
	}
	public Object visitTerminalSymbol(TerminalSymbol terminalsymbol, Object
arg, int line){
		terminalsymbol.terminal.visit(this,null); return null;
	}
	public Object visitNonTerminalSymbol(NonTerminalSymbol nonterminalsymbol,
Object arg, int line){
		nonterminalsymbol.nonTerminal.visit(this,null); return null;
	}
	public Object visitNonTerminal(NonTerminal nonterminal, Object arg, int
line){
//		System.out.println("*NT*"+nonterminal+"*NT*");
		nonTerminalSet.add(nonterminal.spelling);

		return null;
	}
	public Object visitTerminal(Terminal terminal, Object arg, int line){
//		System.out.println("*T*"+terminal+"*T*");
		return null;
	}

	public static void main(String[] args) throws Exception{
		Parser parser = new Parser();
		Grammar grammar = parser.parse();
//		new Display(grammar);
		NonTerminalChecker check = new NonTerminalChecker();
		check.check(grammar);
		System.out.println("The program is terminated normally.");

		System.out.println("*****");
		NonTerminalChecker nonTerminalChecker = new NonTerminalChecker();
		nonTerminalChecker.check(grammar);


	System.out.println("definedNonTerminalSet:"+nonTerminalChecker.getDefinedN
onTerminalSet());

	System.out.println("nonTerminalSet:"+nonTerminalChecker.getUndefinedNonTer
minalSet());

	}



}
```

ebnfast.Ast.java

```
package ebnfast;
```

```
public abstract class Ast{
      public abstract Object visit(Visitor v, Object arg);
}
```

ebnfast.Checker.java

```
package ebnfast;
import ebnfscanner.*;
import ebnfparser.*;

import java.util.*;
public class Checker implements Visitor{
      Set terminalSet = new HashSet();
      public void check(Grammar grammar){
            grammar.visit(this, null);
      }
      public Object visitGrammar(Grammar grammar, Object arg, int line){
            grammar.productionRuleList.visit(this,null); return null;
      }
      public Object visitProductionRuleList(ProductionRuleList
productionrulelist, Object arg, int line){
            productionrulelist.productionRule.visit(this,null); if
(productionrulelist.productionRuleListTail!=null)
                  productionrulelist.productionRuleListTail.visit(this,null);
            return null;
      }
      public Object visitProductionRuleListTail(ProductionRuleListTail
productionrulelisttail, Object arg, int line){
            productionrulelisttail.productionRuleList.visit(this,null); if
(productionrulelisttail.productionRuleListTail!=null)

      productionrulelisttail.productionRuleListTail.visit(this,null);
            return null;
      }
      public Object visitProductionRule(ProductionRule productionrule, Object
arg, int line){
            System.out.println("ProductionRule:"+productionrule);
            productionrule.nonTerminal.visit(this,null);
productionrule.definition.visit(this,null); return null;
      }
      public Object visitDefinition(Definition definition, Object arg, int
line){
//          System.out.println("Definition:"+definition);

            definition.definitionSequence.visit(this,null);
            if (definition.definitionTail!=null)
                  definition.definitionTail.visit(this,null);
            return null;
      }
      public Object visitDefinitionTail(DefinitionTail definitiontail, Object
arg, int line){
            definitiontail.definitionSequence.visit(this,null); if
(definitiontail.definitionTail!=null)
                  definitiontail.definitionTail.visit(this,null);
            return null;
```

```
	}
	public Object visitDefinitionSequence(DefinitionSequence
definitionsequence, Object arg, int line){
		definitionsequence.definitionBody.visit(this,null); if
(definitionsequence.definitionSequence!=null)
			definitionsequence.definitionSequence.visit(this,null);
		return null;
	}
	public Object visitDefinitionBody(DefinitionBody definitionbody, Object
arg, int line){
		return null;
	}
	public Object visitSymbolDefinition(SymbolDefinition symboldefinition,
Object arg, int line){
		symboldefinition.symbol.visit(this,null); return null;
	}
	public Object visitGroupingDefinition(GroupingDefinition
groupingdefinition, Object arg, int line){
		groupingdefinition.definition.visit(this,null); return null;
	}
	public Object visitZeroOrOneDefinition(ZeroOrOneDefinition
zerooronedefinition, Object arg, int line){
		zerooronedefinition.definition.visit(this,null); return null;
	}
	public Object visitZeroOrMoreDefinition(ZeroOrMoreDefinition
zeroormoredefinition, Object arg, int line){
		zeroormoredefinition.definition.visit(this,null); return null;
	}
	public Object visitSymbol(Symbol symbol, Object arg, int line){
		return null;
	}
	public Object visitTerminalSymbol(TerminalSymbol terminalsymbol, Object
arg, int line){
		terminalsymbol.terminal.visit(this,null); return null;
	}
	public Object visitNonTerminalSymbol(NonTerminalSymbol nonterminalsymbol,
Object arg, int line){
		nonterminalsymbol.nonTerminal.visit(this,null); return null;
	}
	public Object visitNonTerminal(NonTerminal nonterminal, Object arg, int
line){
//		System.out.println("*NT*"+nonterminal+"*NT*");
		return null;
	}
	public Object visitTerminal(Terminal terminal, Object arg, int line){
//		System.out.println("*T*"+terminal+"*T*");
		terminalSet.add(terminal.spelling);
		return null;
	}

	public static void main(String[] args) throws Exception{
		Parser parser = new Parser();
		Grammar grammar = parser.parse();
		new Display(grammar);
		Checker check = new Checker();
		check.check(grammar);
		System.out.println("The program is terminated normally.");
```

```
            Iterator iterator = check.terminalSet.iterator();
            while (iterator.hasNext()){
                  String terminal = (String) iterator.next();
                  System.out.println(terminal);

            }
      }
}
```

ebnfast.Definition.java

```
package ebnfast;
public class Definition{
      public DefinitionSequence definitionSequence;
      public DefinitionTail definitionTail;
      public int line;

      public Definition(DefinitionSequence definitionSequence, DefinitionTail
definitionTail, int line){
            this.definitionSequence = definitionSequence;
            this.definitionTail = definitionTail;
            this.line = line;
      }

      public String toString(){
            return "Definition[ "+definitionSequence+","+definitionTail+"]";
      }

            public Object visit(Visitor v, Object arg){
      return v.visitDefinition(this, arg, line);
      }

}
```

ebnfast.DefinitionBody.java

```
package ebnfast;
public abstract class DefinitionBody extends Ast{
      public abstract boolean canBeEmpty();
}
```

ebnfast.DefinitionSequence.java

```
package ebnfast;
public class DefinitionSequence{
      public DefinitionBody definitionBody;
      public DefinitionSequence definitionSequence;
      public int line;
      public DefinitionSequence(DefinitionBody definitionBody,
DefinitionSequence definitionSequence, int line){
```

```
            this.definitionBody = definitionBody;
            this.definitionSequence = definitionSequence;
            this.line = line;
      }

      public String toString(){
            return "DefinitionSequence[
"+definitionBody+","+definitionSequence+"]";
      }

            public Object visit(Visitor v, Object arg){
      return v.visitDefinitionSequence(this, arg, line);
      }

}
```

ebnfast.DefinitionTail.java

```
package ebnfast;
public class DefinitionTail{
      public DefinitionSequence definitionSequence;
      public DefinitionTail definitionTail;
      public int line;

      public DefinitionTail(DefinitionSequence definitionSequence,
DefinitionTail definitionTail, int line){
            this.definitionSequence = definitionSequence;
            this.definitionTail = definitionTail;
            this.line = line;
      }

      public String toString(){
            return "DefinitionTail[ "+definitionSequence+","+definitionTail+"]";
      }

            public Object visit(Visitor v, Object arg){
      return v.visitDefinitionTail(this, arg, line);
      }

}
```

ebnfast.Display.java

```
package ebnfast;
public class Display{
      final private int BLANKS = 1;
      public Display(Object p){
            display(p, 0);
      }
      private static void spaces(int count){
              for(int i = 1; i <= count; i++)
                        System.out.print(" ");
      }
```

```
      private void display(Object p, int count){
            if(p == null) return;
            String s = p.getClass().getName();
            spaces(count);
            System.out.println(s);
            if(s.equals("ebnfast.Grammar")){
                  display(((Grammar)p).productionRuleList, count+BLANKS);
            }
            if(s.equals("ebnfast.ProductionRuleList")){
                  display(((ProductionRuleList)p).productionRule, count+BLANKS);
                  display(((ProductionRuleList)p).productionRuleListTail,
count+BLANKS);
            }
            if(s.equals("ebnfast.ProductionRuleListTail")){
                  display(((ProductionRuleListTail)p).productionRuleList,
count+BLANKS);
                  display(((ProductionRuleListTail)p).productionRuleListTail,
count+BLANKS);
            }
            if(s.equals("ebnfast.ProductionRule")){
                  display(((ProductionRule)p).nonTerminal, count+BLANKS);
                  display(((ProductionRule)p).definition, count+BLANKS);
            }
            if(s.equals("ebnfast.Definition")){
                  display(((Definition)p).definitionSequence, count+BLANKS);
                  display(((Definition)p).definitionTail, count+BLANKS);
            }
            if(s.equals("ebnfast.DefinitionTail")){
                  display(((DefinitionTail)p).definitionSequence, count+BLANKS);
                  display(((DefinitionTail)p).definitionTail, count+BLANKS);
            }
            if(s.equals("ebnfast.DefinitionSequence")){
                  display(((DefinitionSequence)p).definitionBody, count+BLANKS);
                  display(((DefinitionSequence)p).definitionSequence,
count+BLANKS);
            }
            if(s.equals("ebnfast.DefinitionBody")){
            }
            if(s.equals("ebnfast.SymbolDefinition")){
                  display(((SymbolDefinition)p).symbol, count+BLANKS);
            }
            if(s.equals("ebnfast.GroupingDefinition")){
                  display(((GroupingDefinition)p).definition, count+BLANKS);
            }
            if(s.equals("ebnfast.ZeroOrOneDefinition")){
                  display(((ZeroOrOneDefinition)p).definition, count+BLANKS);
            }
            if(s.equals("ebnfast.ZeroOrMoreDefinition")){
                  display(((ZeroOrMoreDefinition)p).definition, count+BLANKS);
            }
            if(s.equals("ebnfast.Symbol")){
            }
            if(s.equals("ebnfast.TerminalSymbol")){
                  display(((TerminalSymbol)p).terminal, count+BLANKS);
            }
            if(s.equals("ebnfast.NonTerminalSymbol")){
                  display(((NonTerminalSymbol)p).nonTerminal, count+BLANKS);
```

```
            }
            if(s.equals("ebnfast.NonTerminal")){
                  spaces(count);
                  System.out.println("***"+((NonTerminal)p).spelling+"***");
            }
            if(s.equals("ebnfast.Terminal")){
                  spaces(count);
                  System.out.println("***"+((Terminal)p).spelling+"***");
            }
      }
}
```

ebnfast.Grammar.java

```
package ebnfast;
public class Grammar{
      public ProductionRuleList productionRuleList;
      public int line;
      public Grammar(ProductionRuleList productionRuleList, int line){
            this.productionRuleList = productionRuleList;
            this.line = line;
      }

      public String toString(){
            return "Grammar[ "+productionRuleList+"]";
      }

            public Object visit(Visitor v, Object arg){
      return v.visitGrammar(this, arg, line);
      }

}
```

ebnfast.GroupingDefinition.java

```
package ebnfast;
public class GroupingDefinition extends DefinitionBody{
      public Definition definition;
      public int line;

      public GroupingDefinition(Definition definition, int line){
            this.definition = definition;
            this.line = line;
      }

      public String toString(){
            return "GroupingDefinition[ "+definition+"]";
      }

            public Object visit(Visitor v, Object arg){
      return v.visitGroupingDefinition(this, arg, line);
      }
```

```
	public boolean canBeEmpty(){
		return false;
	}

}
```

ebnfast.HtmlDisplay.java

```
package ebnfast;
import java.io.*;
public class HtmlDisplay{
	final private static int BLANKS = 2;
	public HtmlDisplay(Object p){
	}

	public static void display(Grammar grammar, Writer out) throws
IOException{
		display(grammar, out, 0);
	}

	private static void spaces(int count, Writer out) throws IOException{
		  for(int i = 1; i <= count; i++)
			    out.write(" ");
	}
	private static void display(Object p, Writer out, int count) throws
IOException{
		if(p == null) return;
		String s = p.getClass().getName();
		spaces(count,out);
		out.write(s+"<br/>");
		if(s.equals("ebnfast.Grammar")){
			display(((Grammar)p).productionRuleList,out,  count+BLANKS);
		}
		if(s.equals("ebnfast.ProductionRuleList")){
			display(((ProductionRuleList)p).productionRule, out,
count+BLANKS);
			display(((ProductionRuleList)p).productionRuleListTail, out,
count+BLANKS);
		}
		if(s.equals("ebnfast.ProductionRuleListTail")){
			display(((ProductionRuleListTail)p).productionRuleList, out,
count+BLANKS);
			display(((ProductionRuleListTail)p).productionRuleListTail,
out, count+BLANKS);
		}
		if(s.equals("ebnfast.ProductionRule")){
			display(((ProductionRule)p).nonTerminal, out, count+BLANKS);
			display(((ProductionRule)p).definition, out, count+BLANKS);
		}
		if(s.equals("ebnfast.Definition")){
			display(((Definition)p).definitionSequence, out,
count+BLANKS);
			display(((Definition)p).definitionTail, out, count+BLANKS);
		}
		if(s.equals("ebnfast.DefinitionTail")){
```

```
                  display(((DefinitionTail)p).definitionSequence, out,
count+BLANKS);
                  display(((DefinitionTail)p).definitionTail, out,
count+BLANKS);
            }
            if(s.equals("ebnfast.DefinitionSequence")){
                  display(((DefinitionSequence)p).definitionBody, out,
count+BLANKS);
                  display(((DefinitionSequence)p).definitionSequence, out,
count+BLANKS);
            }
            if(s.equals("ebnfast.DefinitionBody")){
            }
            if(s.equals("ebnfast.SymbolDefinition")){
                  display(((SymbolDefinition)p).symbol, out, count+BLANKS);
            }
            if(s.equals("ebnfast.GroupingDefinition")){
                  display(((GroupingDefinition)p).definition, out,
count+BLANKS);
            }
            if(s.equals("ebnfast.ZeroOrOneDefinition")){
                  display(((ZeroOrOneDefinition)p).definition, out,
count+BLANKS);
            }
            if(s.equals("ebnfast.ZeroOrMoreDefinition")){
                  display(((ZeroOrMoreDefinition)p).definition, out,
count+BLANKS);
            }
            if(s.equals("ebnfast.Symbol")){
            }
            if(s.equals("ebnfast.TerminalSymbol")){
                  display(((TerminalSymbol)p).terminal, out, count+BLANKS);
            }
            if(s.equals("ebnfast.NonTerminalSymbol")){
                  display(((NonTerminalSymbol)p).nonTerminal, out,
count+BLANKS);
            }
            if(s.equals("ebnfast.NonTerminal")){
                  spaces(count,out);
                  out.write("***"+((NonTerminal)p).spelling+"***<br/>");
            }
            if(s.equals("ebnfast.Terminal")){
                  spaces(count,out);
                  out.write("***"+((Terminal)p).spelling+"***<br/>");
            }
      }
}
```

ebnfast.HtmlLineDisplay.java

```
package ebnfast;
import java.io.*;
public class HtmlLineDisplay{
      final private static int BLANKS = 2;
      public HtmlLineDisplay(Object p){
```

```
	}

	public static void display(Grammar grammar, Writer out) throws
IOException{
		display(grammar, out, 0);
	}

	private static void display(Object p, Writer out, int count) throws
IOException{
		if(p == null) return;
		String s = p.getClass().getName();
		if(s.equals("ebnfast.Grammar")){
			display(((Grammar)p).productionRuleList,out,  count+BLANKS);
		}
		if(s.equals("ebnfast.ProductionRuleList")){
			display(((ProductionRuleList)p).productionRule, out,
count+BLANKS);
			display(((ProductionRuleList)p).productionRuleListTail, out,
count+BLANKS);
		}
		if(s.equals("ebnfast.ProductionRuleListTail")){
			display(((ProductionRuleListTail)p).productionRuleList, out,
count+BLANKS);
			display(((ProductionRuleListTail)p).productionRuleListTail,
out, count+BLANKS);
		}
		if(s.equals("ebnfast.ProductionRule")){
			out.write(((ProductionRule)p).nonTerminal.spelling+" ::=");
			display(((ProductionRule)p).nonTerminal, out, count+BLANKS);
			display(((ProductionRule)p).definition, out, count+BLANKS);
			out.write(".<br/>");
		}
		if(s.equals("ebnfast.Definition")){
			display(((Definition)p).definitionSequence, out,
count+BLANKS);
			display(((Definition)p).definitionTail, out, count+BLANKS);
		}
		if(s.equals("ebnfast.DefinitionTail")){
			out.write(" |");
			display(((DefinitionTail)p).definitionSequence, out,
count+BLANKS);
			display(((DefinitionTail)p).definitionTail, out,
count+BLANKS);
		}
		if(s.equals("ebnfast.DefinitionSequence")){
			out.write(" ");
			display(((DefinitionSequence)p).definitionBody, out,
count+BLANKS);
			display(((DefinitionSequence)p).definitionSequence, out,
count+BLANKS);
		}
		if(s.equals("ebnfast.DefinitionBody")){
		}
		if(s.equals("ebnfast.SymbolDefinition")){
			display(((SymbolDefinition)p).symbol, out, count+BLANKS);
		}
		if(s.equals("ebnfast.GroupingDefinition")){
```

```
                  out.write("(");
                  display(((GroupingDefinition)p).definition, out,
count+BLANKS);
                  out.write(" )");
            }
            if(s.equals("ebnfast.ZeroOrOneDefinition")){
                  out.write("[");
                  display(((ZeroOrOneDefinition)p).definition, out,
count+BLANKS);
                  out.write(" ]");
            }
            if(s.equals("ebnfast.ZeroOrMoreDefinition")){
                  out.write("{");
                  display(((ZeroOrMoreDefinition)p).definition, out,
count+BLANKS);
                  out.write(" }");
            }
            if(s.equals("ebnfast.Symbol")){
            }
            if(s.equals("ebnfast.TerminalSymbol")){
                  out.write("\""+((TerminalSymbol)p).terminal.spelling+"\"");
                  display(((TerminalSymbol)p).terminal, out, count+BLANKS);
            }
            if(s.equals("ebnfast.NonTerminalSymbol")){
                  out.write(((NonTerminalSymbol)p).nonTerminal.spelling);
                  display(((NonTerminalSymbol)p).nonTerminal, out,
count+BLANKS);
            }
            if(s.equals("ebnfast.NonTerminal")){
            }
            if(s.equals("ebnfast.Terminal")){
            }
      }


      public static void main(String args[]) throws Exception{
            ebnfparser.Parser parser = new ebnfparser.Parser();
            Grammar grammar = parser.parse();
            OutputStream outStream = new ByteArrayOutputStream();
            Writer out = new PrintWriter(outStream);
            HtmlLineDisplay.display(grammar,out);
            out.flush();
            System.out.println(outStream.toString());

      }

}
```

ebnfast.NonTerminal.java

```
package ebnfast;
public class NonTerminal{
      public String spelling;
      public int line;
```

```
	public NonTerminal(String spelling, int line){
		System.out.println("===NT===>"+spelling);
		this.spelling = spelling;
		this.line = line;
	}

	public String toString(){
		return "NonTerminal[ \""+spelling+"\" ]";
	}

		public Object visit(Visitor v, Object arg){
	return v.visitNonTerminal(this, arg, line);
	}

}
```

ebnfast.NonTerminalSymbol.java

```
package ebnfast;
public class NonTerminalSymbol extends Symbol{
	public NonTerminal nonTerminal;
	public int line;

	public NonTerminalSymbol(NonTerminal nonTerminal, int line){
		this.nonTerminal = nonTerminal;
		this.line = line;
	}

	public String toString(){
		return "NonTerminalSymbol[ "+nonTerminal+"]";
	}

		public Object visit(Visitor v, Object arg){
	return v.visitNonTerminalSymbol(this, arg, line);
	}
	public String getSpelling(){
		return nonTerminal.spelling;
	}

}
```

ebnfast.ProductionRule.java

```
package ebnfast;
public class ProductionRule{
	public NonTerminal nonTerminal;
	public Definition definition;
	public int line;

	public ProductionRule(NonTerminal nonTerminal, Definition definition, int
line){
		this.nonTerminal = nonTerminal;
		this.definition = definition;
```

```
            this.line = line;
      }

      public String toString(){
            return "ProductionRule[ "+nonTerminal+","+definition+"]";
      }

            public Object visit(Visitor v, Object arg){
      return v.visitProductionRule(this, arg, line);
      }

}
```

ebnfast.ProductionRuleList.java

```
package ebnfast;
public class ProductionRuleList{
      public ProductionRule productionRule;
      public ProductionRuleListTail productionRuleListTail;
      public int line;
      public ProductionRuleList(ProductionRule productionRule,
ProductionRuleListTail productionRuleListTail, int line){
            this.productionRule = productionRule;
            this.productionRuleListTail = productionRuleListTail;
            this.line = line;
      }

      public String toString(){
            return "ProductionRuleList[
"+productionRule+","+productionRuleListTail+"]";
      }

            public Object visit(Visitor v, Object arg){
      return v.visitProductionRuleList(this, arg, line);
      }

}
```

ebnfast.ProductionRuleListTail.java

```
package ebnfast;
public class ProductionRuleListTail{
      public ProductionRuleList productionRuleList;
      public ProductionRuleListTail productionRuleListTail;
      public int line;
      public ProductionRuleListTail(ProductionRuleList productionRuleList,
ProductionRuleListTail productionRuleListTail, int line){
            this.productionRuleList = productionRuleList;
            this.productionRuleListTail = productionRuleListTail;
            this.line = line;
      }

      public String toString(){
```

```
            return "ProductionRuleListTail[
"+productionRuleList+","+productionRuleListTail+"]";
      }

            public Object visit(Visitor v, Object arg){
      return v.visitProductionRuleListTail(this, arg, line);
      }

}
```

ebnfast.Symbol.java

```
package ebnfast;
public abstract class Symbol extends Ast{
      public abstract String getSpelling();
}
```

ebnfast.SymbolDefinition.java

```
package ebnfast;
public class SymbolDefinition extends DefinitionBody{
      public Symbol symbol;
      public int line;

      public SymbolDefinition(Symbol symbol, int line){
            this.symbol = symbol;
            this.line = line;
      }

      public String toString(){
            return "SymbolDefinition[ "+symbol+"]";
      }

            public Object visit(Visitor v, Object arg){
      return v.visitSymbolDefinition(this, arg, line);
      }

      public boolean canBeEmpty(){
            return false;
      }

}
```

ebnfast.Terminal.java

```
package ebnfast;
public class Terminal{
      public String spelling;
      public int line;

      public Terminal(String spelling, int line){
```

```
            System.out.println("===T===>"+spelling);
            this.spelling = spelling;
            this.line = line;
      }

      public String toString(){
            return "Terminal[ \""+spelling+"\" ]";
      }

            public Object visit(Visitor v, Object arg){
      return v.visitTerminal(this, arg, line);
      }

}
```

ebnfast.TerminalSymbol.java

```
package ebnfast;
public class TerminalSymbol extends Symbol{
      public Terminal terminal;
      public int line;

      public TerminalSymbol(Terminal terminal, int line){
            this.terminal = terminal;
            this.line = line;
      }

      public String toString(){
            return "TerminalSymbol[ "+terminal+"]";
      }

            public Object visit(Visitor v, Object arg){
      return v.visitTerminalSymbol(this, arg, line);
      }
      public String getSpelling(){
            return terminal.spelling;
      }

}
```

ebnfast.Visitor.java

```
package ebnfast;
public interface Visitor{
      public Object visitGrammar(Grammar grammar, Object arg, int line);
      public Object visitProductionRuleList(ProductionRuleList
productionrulelist, Object arg, int line);
      public Object visitProductionRule(ProductionRule productionrule, Object
arg, int line);
      public Object visitProductionRuleListTail(ProductionRuleListTail
productionrulelisttail, Object arg, int line);
      public Object visitNonTerminal(NonTerminal nonterminal, Object arg, int
line);
```

```
      public Object visitDefinition(Definition definition, Object arg, int
line);
      public Object visitDefinitionSequence(DefinitionSequence
definitionsequence, Object arg, int line);
      public Object visitDefinitionTail(DefinitionTail definitiontail, Object
arg, int line);
      public Object visitDefinitionBody(DefinitionBody definitionbody, Object
arg, int line);
      public Object visitSymbolDefinition(SymbolDefinition symboldefinition,
Object arg, int line);
      public Object visitSymbol(Symbol symbol, Object arg, int line);
      public Object visitGroupingDefinition(GroupingDefinition
groupingdefinition, Object arg, int line);
      public Object visitZeroOrOneDefinition(ZeroOrOneDefinition
zerooronedefinition, Object arg, int line);
      public Object visitZeroOrMoreDefinition(ZeroOrMoreDefinition
zeroormoredefinition, Object arg, int line);
      public Object visitTerminalSymbol(TerminalSymbol terminalsymbol, Object
arg, int line);
      public Object visitTerminal(Terminal terminal, Object arg, int line);
      public Object visitNonTerminalSymbol(NonTerminalSymbol nonterminalsymbol,
Object arg, int line);

}
```

ebnfast.ZeroOrMoreDefinition.java

```
package ebnfast;
public class ZeroOrMoreDefinition extends DefinitionBody{
      public Definition definition;
      public int line;

      public ZeroOrMoreDefinition(Definition definition, int line){
            this.definition = definition;
            this.line = line;
      }

      public String toString(){
            return "ZeroOrMoreDefinition[ "+definition+"]";
      }

            public Object visit(Visitor v, Object arg){
      return v.visitZeroOrMoreDefinition(this, arg, line);
      }

      public boolean canBeEmpty(){
            return true;
      }
}
```

ebnfast.ZeroOrOneDefinition.java

```
package ebnfast;
```

```
public class ZeroOrOneDefinition extends DefinitionBody{
	public Definition definition;
	public int line;

	public ZeroOrOneDefinition(Definition definition, int line){
		this.definition = definition;
		this.line = line;
	}

	public String toString(){
		return "ZeroOrOneDefinition[ "+definition+"]";
	}

		public Object visit(Visitor v, Object arg){
	return v.visitZeroOrOneDefinition(this, arg, line);
	}

	public boolean canBeEmpty(){
		return true;
	}

}
```

ebnfparser.GrammarChecker.java

```
package ebnfparser;
import ebnfast.*;
import ebnfscanner.*;
import ebnfparser.*;
import common.*;

import java.util.*;
public class GrammarChecker implements Visitor{
	//store all production rules to access them backwards Set
definedTerminalSet = new HashSet();
	Set undefinedTerminalSet = new HashSet();
	Set usedTerminalSet = new HashSet();

	public void check(Grammar grammar){
		grammar.visit(this, null);
		Iterator usedTerminalsIterator = usedTerminalSet.iterator();
		while (usedTerminalsIterator.hasNext()){
			String usedTerminal = (String) usedTerminalsIterator.next();
			if (!definedTerminalSet.contains(usedTerminal)){
				undefinedTerminalSet.add(usedTerminal);
			}
		}
	}


	public void check(Grammar grammar, GenericToken genericToken){
		grammar.visit(this, null);
		Iterator usedTerminalsIterator = usedTerminalSet.iterator();
		while (usedTerminalsIterator.hasNext()){
			String usedTerminal = (String) usedTerminalsIterator.next();
```

```
                  if (!definedTerminalSet.contains(usedTerminal) &&
genericToken.lookupConstantSpelling(usedTerminal)==null){
                        undefinedTerminalSet.add(usedTerminal);
                  }
            }

      }

      public Object visitGrammar(Grammar grammar, Object arg, int line){
            grammar.productionRuleList.visit(this,null); return null;
      }
      public Object visitProductionRuleList(ProductionRuleList
productionrulelist, Object arg, int line){
            productionrulelist.productionRule.visit(this,null); if
(productionrulelist.productionRuleListTail!=null)
                  productionrulelist.productionRuleListTail.visit(this,null);
            return null;
      }
      public Object visitProductionRuleListTail(ProductionRuleListTail
productionrulelisttail, Object arg, int line){
            productionrulelisttail.productionRuleList.visit(this,null); if
(productionrulelisttail.productionRuleListTail!=null)

      productionrulelisttail.productionRuleListTail.visit(this,null);
            return null;
      }
      public Object visitProductionRule(ProductionRule productionrule, Object
arg, int line){
            definedTerminalSet.add(productionrule.nonTerminal.spelling);
//          productionrule.nonTerminal.visit(this,null);
            productionrule.definition.visit(this,null);
            return null;
      }
      public Object visitDefinition(Definition definition, Object arg, int
line){
//          System.out.println("Definition:"+definition);

            definition.definitionSequence.visit(this,null);
            if (definition.definitionTail!=null)
                  definition.definitionTail.visit(this,null);
            return null;
      }
      public Object visitDefinitionTail(DefinitionTail definitiontail, Object
arg, int line){
            definitiontail.definitionSequence.visit(this,null); if
(definitiontail.definitionTail!=null)
                  definitiontail.definitionTail.visit(this,null);
            return null;
      }
      public Object visitDefinitionSequence(DefinitionSequence
definitionsequence, Object arg, int line){
            definitionsequence.definitionBody.visit(this,null); if
(definitionsequence.definitionSequence!=null)
                  definitionsequence.definitionSequence.visit(this,null);
            return null;
      }
```

```
	public Object visitDefinitionBody(DefinitionBody definitionbody, Object
arg, int line){
		return null;
	}
	public Object visitSymbolDefinition(SymbolDefinition symboldefinition,
Object arg, int line){
		symboldefinition.symbol.visit(this,null); return null;
	}
	public Object visitGroupingDefinition(GroupingDefinition
groupingdefinition, Object arg, int line){
		groupingdefinition.definition.visit(this,null); return null;
	}
	public Object visitZeroOrOneDefinition(ZeroOrOneDefinition
zerooronedefinition, Object arg, int line){
		zerooronedefinition.definition.visit(this,null); return null;
	}
	public Object visitZeroOrMoreDefinition(ZeroOrMoreDefinition
zeroormoredefinition, Object arg, int line){
		zeroormoredefinition.definition.visit(this,null); return null;
	}
	public Object visitSymbol(Symbol symbol, Object arg, int line){
		return null;
	}
	public Object visitTerminalSymbol(TerminalSymbol terminalsymbol, Object
arg, int line){
		terminalsymbol.terminal.visit(this,null); return null;
	}
	public Object visitNonTerminalSymbol(NonTerminalSymbol nonterminalsymbol,
Object arg, int line){
		nonterminalsymbol.nonTerminal.visit(this,null); return null;
	}
	public Object visitNonTerminal(NonTerminal nonterminal, Object arg, int
line){
		//System.out.println("*NT*"+nonterminal+"*NT*");
		usedTerminalSet.add(nonterminal.spelling);
		return null;
	}
	public Object visitTerminal(Terminal terminal, Object arg, int line){
		//System.out.println("*T*"+terminal+"*T*");
		return null;
	}

	public static void main(String[] args) throws Exception{
		Parser parser = new Parser();
		Grammar grammar = parser.parse();
		new Display(grammar);
		GrammarChecker check = new GrammarChecker();
		check.check(grammar);
		System.out.println("*****");
		System.out.println("definedTerminalSet:"+check.definedTerminalSet);
		System.out.println("usedTerminalSet:"+check.usedTerminalSet);

	System.out.println("undefinedTerminalSet:"+check.undefinedTerminalSet);
		System.out.println("The program is terminated normally.");

	}
```

```
    /**
     * @return
     */
    public Set getDefinedTerminalSet() {
        return definedTerminalSet;
    }

    /**
     * @return
     */
    public Set getUndefinedTerminalSet() {
        return undefinedTerminalSet;
    }

    /**
     * @return
     */
    public Set getUsedTerminalSet() {
        return usedTerminalSet;
    }

}
```

ebnfparser.Parser.java

```
package ebnfparser;
import ebnfast.*;
import ebnfscanner.*;
import common.Error;
import common.SourceFile;

import java.io.*;
public class Parser{
  private Token currentToken;
  Scanner scanner;

  private void accept(byte expectedKind) throws ScannerException{
    if(currentToken.kind == expectedKind){
            System.out.println("Accepted: "+currentToken);
      currentToken = scanner.scan();
      }
    else
      throw new ScannerException("Syntax error: " + currentToken + " is not
expected.  Expected "+expectedKind, currentToken.line);
  }

  private void acceptIt() throws ScannerException{
      System.out.println("Accepted: "+currentToken);
    currentToken = scanner.scan();
  }


  public Grammar parse() throws ScannerException, ParserException{
```

```
    SourceFile sourceFile = new SourceFile();
    scanner = new Scanner(sourceFile.openFile());
    currentToken = scanner.scan();
      Grammar grammar = parseGrammar();
    if(currentToken.kind != Token.EOT)
            throw new ScannerException("Syntax error: Redundant characters at
the end of program.", currentToken.line);
    return grammar;
  }

  public Grammar parse(String grammarText) throws ScannerException,
ParserException{
      //InputSource grammarInputSource =
      StringReader grammarReader = new StringReader(grammarText);
      BufferedReader grammarBufferedReader = new BufferedReader(grammarReader);
      scanner = new Scanner(grammarBufferedReader);
      currentToken = scanner.scan();
      Grammar grammar = parseGrammar();
      if(currentToken.kind != Token.EOT)
            throw new ScannerException("Syntax error: Redundant characters at
the end of program.", currentToken.line);
      return grammar;
  }



//Grammar         ::= ProductionRuleList .
private Grammar parseGrammar() throws ScannerException,ParserException{
      System.out.println(":"+currentToken.line+":"+currentToken.spelling+":"+cur
rentToken.kind+":Grammar");
      ProductionRuleList prl = parseProductionRuleList();
      return new Grammar(prl,currentToken.line);
}

//ProductionRuleList    ::= ProductionRule {ProductionRuleList} .
private ProductionRuleList parseProductionRuleList() throws
ScannerException,ParserException{
      System.out.println(":"+currentToken.line+":"+currentToken.spelling+":"+cur
rentToken.kind+":ProductionRuleList");
      ProductionRule pr = parseProductionRule();
      ProductionRuleListTail prlt = null;
      ProductionRuleListTail current = null;
      ProductionRuleListTail prev = null;

      while (currentToken.kind != Token.EOT){
            ProductionRuleList prl = parseProductionRuleList();
            current = new ProductionRuleListTail(prl,null,currentToken.line);
            if (prlt==null)
                  prlt = current;
            else
                  prev.productionRuleListTail = current;
            prev = current;

      }

      return new ProductionRuleList(pr,prlt,currentToken.line);
```

```
}

//ProductionRule  ::= NonTerminal "::=" Definition ".".
private ProductionRule parseProductionRule() throws
ScannerException,ParserException{
      System.out.println(":"+currentToken.line+":"+currentToken.spelling+":"+cur
rentToken.kind+":***ProductionRule***");
      NonTerminal nt = parseNonTerminal();
      accept(Token.RULE);
      Definition d = parseDefinition();
      accept(Token.ENDRULE);

      return new ProductionRule(nt,d,currentToken.line);
}


//Definition                  ::= DefinitionSequence DefinitionTail .
private Definition parseDefinition() throws ScannerException,ParserException{
      System.out.println(":"+currentToken.line+":"+currentToken.spelling+":"+cur
rentToken.kind+":Definition");
      System.out.println("Start Definition");
      DefinitionSequence ds = parseDefinitionSequence();
      DefinitionTail dt = null;
      DefinitionTail current = null;
      DefinitionTail prev = null;

      while (currentToken.kind == Token.ALTERNATIVE){
            acceptIt();
            DefinitionSequence ds2 = parseDefinitionSequence();
            current = new DefinitionTail(ds2,null,currentToken.line);
            if (dt==null)
                  dt = current;
            else
                  prev.definitionTail = current;
            prev = current;

      }

      System.out.println("End Definition");
      return new Definition(ds,dt,currentToken.line);
}


//DefinitionSequence ::= DefinitionBody DefinitionSequenceTail .
private DefinitionSequence parseDefinitionSequence() throws
ScannerException,ParserException{
      System.out.println(":"+currentToken.line+":"+currentToken.spelling+":"+cur
rentToken.kind+":DefinitionSequence");
      DefinitionSequence ds = null;
      DefinitionBody db = parseDefinitionBody();
      DefinitionSequence ds2 = null;
      DefinitionSequence current = null;
      DefinitionSequence prev = null;

      while (currentToken.kind==Token.NONTERMINAL ||
currentToken.kind==Token.TERMINAL || currentToken.kind==Token.PARENOPEN ||
```

```
currentToken.kind==Token.BRACKETOPEN ||
currentToken.kind==Token.CURLYBRACKETOPEN){
            //acceptIt();
            DefinitionBody db2 = parseDefinitionBody();
            current = new DefinitionSequence(db2,null,currentToken.line);
            if (ds2==null)
                  ds2 = current;
            else
                  prev.definitionSequence = current;
            prev = current;

      }

      return new DefinitionSequence(db,ds2,currentToken.line);
}



//DefinitionBody ::=    Symbol {Definition} | "(" Definition ")" | "["
Definition "]" |"{" Definition "}".
private DefinitionBody parseDefinitionBody() throws
ScannerException,ParserException{
      System.out.println(":"+currentToken.line+":"+currentToken.spelling+":"+cur
rentToken.kind+":DefinitionBody");
      DefinitionBody db = null;
      if (currentToken.kind==Token.NONTERMINAL ||
currentToken.kind==Token.TERMINAL){
            System.out.println("Start SymbolDefinition");
            Symbol s = parseSymbol();
            /*
            SymbolDefinitionTail sdt = null;
            SymbolDefinitionTail current = null;
            SymbolDefinitionTail prev = null;
            while (currentToken.kind==Token.NONTERMINAL ||
currentToken.kind==Token.TERMINAL || currentToken.kind==Token.PARENOPEN ||
currentToken.kind==Token.BRACKETOPEN ||
currentToken.kind==Token.CURLYBRACKETOPEN){
                  //acceptIt();
                  Definition d = parseDefinition();
                  current = new SymbolDefinitionTail(d,null,currentToken.line);
                  if (sdt==null)
                        sdt = current;
                  else
                        prev.symbolDefinitionTail = current;
                  prev = current;

            }
            */
            System.out.println("End SymbolDefinition");
            return new SymbolDefinition(s,currentToken.line);
      }

      if (currentToken.kind==Token.PARENOPEN){
            System.out.println("Start GroupingDefinition");
            acceptIt();
            Definition d = parseDefinition();
            accept(Token.PARENCLOSE);
```

```
            System.out.println("End GroupingDefinition");
            return new GroupingDefinition(d,currentToken.line);
      }

      if (currentToken.kind==Token.BRACKETOPEN){
            System.out.println("Start ZeroOrOneDefinition");
            acceptIt();
            Definition d = parseDefinition();
            accept(Token.BRACKETCLOSE);

            System.out.println("End ZeroOrOneDefinition");
            return new ZeroOrOneDefinition(d,currentToken.line);
      }

      if (currentToken.kind==Token.CURLYBRACKETOPEN){
            System.out.println("Start ZeroOrMoreDefinition");
            acceptIt();
            Definition d = parseDefinition();
            accept(Token.CURLYBRACKETCLOSE);

            System.out.println("End ZeroOrMoreDefinition");
            return new ZeroOrMoreDefinition(d,currentToken.line);
      }


      throw new ParserException("DefinitionBody error",currentToken.line);
}


//Symbol          ::= Terminal | NonTerminal .
private Symbol parseSymbol() throws ScannerException,ParserException{
      System.out.println(":"+currentToken.line+":"+currentToken.spelling+":"+cur
rentToken.kind+":Symbol");
      if (currentToken.kind==Token.NONTERMINAL){
            NonTerminal nt = parseNonTerminal();
            return new NonTerminalSymbol(nt,currentToken.line);
      }
      else if (currentToken.kind==Token.TERMINAL){
            Terminal t = parseTerminal();
            return new TerminalSymbol(t,currentToken.line);
      }
      else
            throw new ParserException("Symbol expected instead of
"+currentToken+".", currentToken.line);

}

//NonTerminal           ::= NONTERMINAL .
private NonTerminal parseNonTerminal() throws ScannerException,ParserException{
      System.out.println(":"+currentToken.line+":"+currentToken.spelling+":"+cur
rentToken.kind+":NonTerminal");
      if (currentToken.kind!=Token.NONTERMINAL){
            throw new ParserException("Expected
NonTerminal.",currentToken.line);
      }
      else
```

```
      {
            NonTerminal nt = new
NonTerminal(currentToken.spelling,currentToken.line);
            acceptIt();
            return nt;
      }

}

//Terminal        ::= " String " .
private Terminal parseTerminal() throws ScannerException,ParserException{
      System.out.println(":"+currentToken.line+":"+currentToken.spelling+":"+cur
rentToken.kind+":Terminal");
      if (currentToken.kind!=Token.TERMINAL){
            throw new ParserException("Expected Terminal.",currentToken.line);
      }
      else
      {
            Terminal t = new Terminal(currentToken.spelling,currentToken.line);
            acceptIt();
            return t;
      }

}

  public static void main(String[] args) throws Exception{
      System.out.println("Starting");
    Parser parser = new Parser();
    Grammar grammar = parser.parse("L::= \"hi\".");
      System.out.println("**************************");
      System.out.println(grammar);
      System.out.println("**************************");
      new Display(grammar);
      System.out.println("Stopping");
  }

}
```

ebnfparser.ParserException.java

```
/*
 * Created on Mar 15, 2004
 */
package ebnfparser;

/**
 * @author jfoure
 */
public class ParserException extends Exception {
      protected int line;
      public ParserException(String message, int line){
            super(message);
            setLine(line);
      }
```

```
      /**
       * @return
       */
      public int getLine() {
            return line;
      }

      /**
       * @param i
       */
      public void setLine(int i) {
            line = i;
      }


}
```

ebnfscanner.Scanner.java

```
package ebnfscanner;
import java.io.*;
import common.Error;
import common.SourceFile;

public class Scanner{
  private char currentChar;
  private byte currentKind;
  private StringBuffer currentSpelling;
  private BufferedReader inFile;
  private static int line = 1;

  public Scanner(BufferedReader inFile){
    this.inFile = inFile;
      try{
        int i = this.inFile.read();
        if(i == -1) //end of file
            currentChar = '\u0000';
        else
            currentChar = (char)i;
      }
      catch(IOException e){
            System.out.println(e);
      }
  }

  private void takeIt(){
    currentSpelling.append(currentChar);
    try{
      int i = inFile.read();
      if(i == -1) //end of file
        currentChar = '\u0000';
      else
        currentChar = (char)i;
    }
    catch(IOException e){
```

```
        System.out.println(e);
    }
  }

  private void discard(){
    try{
      int i = inFile.read();
      if(i == -1) //end of file
        currentChar = '\u0000';
      else
        currentChar = (char)i;
    }
    catch(IOException e){
        System.out.println(e);
    }
  }

  private boolean isDigit(char c){
    return '0' <= c && c <= '9';
  }

  private boolean isLetter(char c){
    return ('a' <= c && c <= 'z') || ('A' <= c && c <= 'Z');
  }

  private boolean isGraphic(char c){
    return c!='"' && c!='`' && c!='”' && (c == '\t' ||(' ' <= c && c <= '~'));
  }

  private boolean isBlank(char c){
      return (c == ' ' || c == '\n' || c == '\r' || c == '\t');
  }

  private byte scanToken() throws ScannerException{
//System.out.println("scanToken: "+currentChar);
    switch(currentChar){

        case 'a': case 'b': case 'c': case 'd': case 'e': case 'f': case 'g':
case 'h': case 'i':
      case 'j': case 'k': case 'l': case 'm': case 'n': case 'o': case 'p': case
'q': case 'r':
      case 's': case 't': case 'u': case 'v': case 'w': case 'x': case 'y': case
'z':
      case 'A': case 'B': case 'C': case 'D': case 'E': case 'F': case 'G': case
'H': case 'I':
      case 'J': case 'K': case 'L': case 'M': case 'N': case 'O': case 'P': case
'Q': case 'R':
      case 'S': case 'T': case 'U': case 'V': case 'W': case 'X': case 'Y': case
'Z':
        takeIt();
        while(isLetter(currentChar) || isDigit(currentChar))
          takeIt();
        return Token.NONTERMINAL;

        case '"': case '`': case '”':
            discard();
            while(isGraphic(currentChar))
```

```
            takeIt();
          if (currentChar=='"' || currentChar=='‟' || currentChar=='”')
              discard();
          return Token.TERMINAL;
      case ':':
          takeIt();
          if(currentChar != ':'){
              new Error("wrong token [" + currentChar+"]", line);
              return Token.EOT;
          }
          takeIt();
          if(currentChar != '='){
              new Error("wrong token [" + currentChar+"]", line);
              return Token.EOT;
          }
          takeIt();
          return Token.RULE;
      case '.':
          takeIt();
          return Token.ENDRULE;
      case '|':
          takeIt();
          return Token.ALTERNATIVE;

          case '(':
                takeIt();
                return Token.PARENOPEN;
          case ')':
                takeIt();
                return Token.PARENCLOSE;
          case '[':
                takeIt();
                return Token.BRACKETOPEN;
          case ']':
                takeIt();
                return Token.BRACKETCLOSE;
          case '{':
                takeIt();
                return Token.CURLYBRACKETOPEN;
          case '}':
                takeIt();
                return Token.CURLYBRACKETCLOSE;
      case '\u0000':
          return Token.EOT;
    default:
      throw new ScannerException("wrong token [" + currentChar+"]", line);
  }



}

private void scanSeparator(){
  switch(currentChar){
    case '!':
      discard();
      while(isGraphic(currentChar))
```

```
          discard();
        if(currentChar == '\r')
          discard();
        discard();
        line++;
        break;
      case ' ': case '\n': case '\r': case '\t':
        if(currentChar == '\n')
          line++;
        discard();
        while(isBlank(currentChar)){
            if(currentChar == '\n')
              line++;
            discard();
        }
    }
  }

  public Token scan() throws ScannerException{
    currentSpelling = new StringBuffer("");
    while(currentChar == '!' || currentChar == ' ' || currentChar == '\n' ||
          currentChar == '\r' || currentChar == '\t')
      scanSeparator();
    currentKind = scanToken();
    return new Token(currentKind, currentSpelling.toString(), line);
  }



  public static void main(String[] args) throws Exception{
    SourceFile sourceFile = new SourceFile();
    Token token;

    System.out.println("Scanner start.");
    Scanner s = new Scanner(sourceFile.openFile());
    do{
      token = s.scan();
      System.out.println("Line: " + token.line + ", spelling = [" +
        token.spelling + "], " + "kind = " + token.kind);
    }while(token.kind != Token.EOT);
      System.out.println("Scanner stop.");
  }
}
```

ebnfscanner.ScannerException.java

```
/*
 * Created on Mar 15, 2004
 */
package ebnfscanner;

/**
 * @author jfoure
 */
public class ScannerException extends Exception {
```

```
      protected int line;
      public ScannerException(String message, int line){
            super(message);
            setLine(line);
      }

      /**
       * @return
       */
      public int getLine() {
            return line;
      }

      /**
       * @param i
       */
      public void setLine(int i) {
            line = i;
      }

}
```

ebnfscanner.Token.java

```
package ebnfscanner;
public class Token{
  public byte kind;
  public String spelling;
  public int line;

  public String toString(){
      return "Token ["+kind+", "+spelling+", "+line+"]";
  }

  public Token(byte kind, String spelling, int line){
    this.kind = kind;
    this.spelling = spelling;
    this.line = line;

/*
    if(kind == TERMINAL)
      for(int k = RULETERMINAL; k <= CURLYBRACKETCLOSE; k++)
        if(spelling.equals(spellings[k])){
          this.kind = (byte)k;
          break;
        }
*/

  }

  public final static byte
      TERMINAL    = 0,        //something that has no production rule
    NONTERMINAL = 1,          //has a production rule
      RULE        = 2,        // ::=
      ENDRULE         = 3,          // .
```

```
      ALTERNATIVE = 4,        // |
//    QUOTE       = 5,        // "
      EOT         = 6,
      RULETERMINAL = 7,       //"::="
      ENDRULETERMINAL   = 8,  //"."
      ALTERNATIVETERMINAL = 9,      //|
      PARENOPEN = 10,   //(
      PARENCLOSE = 11,  //)
      BRACKETOPEN = 12, //[
      BRACKETCLOSE      = 13, //]
      CURLYBRACKETOPEN = 14,  //{
      CURLYBRACKETCLOSE = 15; //}


  private final static String[] spellings = {

"<terminal>","<nonterminal>","<rule>","<endrule>","<alternative>","<quote>","<eo
t>",
    "::=",".","|","(",")","[","]","{","}"
  };

}
```

ebnfsets.StarterSet.java

```
/*
 * Created on Feb 15, 2004
 */
package ebnfsets;
import ebnfast.*;
import java.util.*;
/**
 * @author jfoure
 */
public class StarterSet{
      public Set symbolSet = new HashSet();
      boolean canBeEmpty = false;

      /**
       * @return
       */
      public boolean isCanBeEmpty() {
            return canBeEmpty;
      }

      /**
       * @return
       */
      public Set getSymbolSet() {
            return symbolSet;
      }

      public Set getSymbolSpellingSet() {
            Set symbolSpellingSet = new HashSet();
            Iterator iterator = getSymbolSet().iterator();
```

```
                while (iterator.hasNext()){
                      Symbol symbol = (Symbol) iterator.next();
                      if (symbol instanceof NonTerminalSymbol)

      symbolSpellingSet.add(((NonTerminalSymbol)symbol).nonTerminal.spelling);
                      if (symbol instanceof TerminalSymbol)

      symbolSpellingSet.add("\""+((TerminalSymbol)symbol).terminal.spelling+"\""
);
                }
                return      symbolSpellingSet;
      }

      /**
       * @param b
       */
      public void setCanBeEmpty(boolean b) {
            canBeEmpty = b;
      }

      /**
       * @param set
       */
      public void setSymbolSet(Set set) {
            symbolSet = set;
      }

      public void add(Object object){
            symbolSet.add(object);
      }

      public void addAll(Collection collection){
            symbolSet.addAll(collection);
      }
      public void addAll(StarterSet starterSet){
            symbolSet.addAll(starterSet.getSymbolSet());
            if (starterSet.isCanBeEmpty())
                  this.setCanBeEmpty(true);
      }



      public String toString(){
            StringBuffer out = new StringBuffer();
            out.append("StarterSet [ ");
            out.append(canBeEmpty);
            out.append(":");

            Iterator iterator = getSymbolSet().iterator();
            while (iterator.hasNext()){
                  Symbol symbol = (Symbol) iterator.next();
                  if (symbol instanceof NonTerminalSymbol)

      out.append("**"+((NonTerminalSymbol)symbol).nonTerminal.spelling+"**");
                  if (symbol instanceof TerminalSymbol)
                        out.append(((TerminalSymbol)symbol).terminal.spelling);
                  if (iterator.hasNext())
```

```
                        out.append(", ");
                }
                out.append("]");
                return out.toString();
        }
}
```

ebnfsets.StarterSetChecker.java

```
package ebnfsets;
import ebnfast.*;
import ebnfscanner.*;
import ebnfparser.*;

import java.util.*;
public class StarterSetChecker implements Visitor{
        //store all production rules to access them backwards List
productionRuleList = new ArrayList();
        public void check(Grammar grammar){
                grammar.visit(this, null);
        }
        public Object visitGrammar(Grammar grammar, Object arg, int line){
                grammar.productionRuleList.visit(this,null); return null;
        }
        public Object visitProductionRuleList(ProductionRuleList
productionrulelist, Object arg, int line){
                productionrulelist.productionRule.visit(this,null); if
(productionrulelist.productionRuleListTail!=null)
                        productionrulelist.productionRuleListTail.visit(this,null);
                return null;
        }
        public Object visitProductionRuleListTail(ProductionRuleListTail
productionrulelisttail, Object arg, int line){
                productionrulelisttail.productionRuleList.visit(this,null); if
(productionrulelisttail.productionRuleListTail!=null)

        productionrulelisttail.productionRuleListTail.visit(this,null);
                return null;
        }
        public Object visitProductionRule(ProductionRule productionrule, Object
arg, int line){
                System.out.println("ProductionRule:"+productionrule);
                if (productionrule.definition.definitionTail==null)
                        System.out.println("Only 1 alternative.");
                productionRuleList.add(productionrule);

                productionrule.nonTerminal.visit(this,null);
                productionrule.definition.visit(this,null);
                return null;
        }
        public Object visitDefinition(Definition definition, Object arg, int
line){
//              System.out.println("Definition:"+definition);

                definition.definitionSequence.visit(this,null);
```

```
            if (definition.definitionTail!=null)
                  definition.definitionTail.visit(this,null);
            return null;
      }
      public Object visitDefinitionTail(DefinitionTail definitiontail, Object
arg, int line){
            definitiontail.definitionSequence.visit(this,null); if
(definitiontail.definitionTail!=null)
                  definitiontail.definitionTail.visit(this,null);
            return null;
      }
      public Object visitDefinitionSequence(DefinitionSequence
definitionsequence, Object arg, int line){
            definitionsequence.definitionBody.visit(this,null); if
(definitionsequence.definitionSequence!=null)
                  definitionsequence.definitionSequence.visit(this,null);
            return null;
      }
      public Object visitDefinitionBody(DefinitionBody definitionbody, Object
arg, int line){
            return null;
      }
      public Object visitSymbolDefinition(SymbolDefinition symboldefinition,
Object arg, int line){
            symboldefinition.symbol.visit(this,null); return null;
      }
      public Object visitGroupingDefinition(GroupingDefinition
groupingdefinition, Object arg, int line){
            groupingdefinition.definition.visit(this,null); return null;
      }
      public Object visitZeroOrOneDefinition(ZeroOrOneDefinition
zerooronedefinition, Object arg, int line){
            zerooronedefinition.definition.visit(this,null); return null;
      }
      public Object visitZeroOrMoreDefinition(ZeroOrMoreDefinition
zeroormoredefinition, Object arg, int line){
            zeroormoredefinition.definition.visit(this,null); return null;
      }
      public Object visitSymbol(Symbol symbol, Object arg, int line){
            return null;
      }
      public Object visitTerminalSymbol(TerminalSymbol terminalsymbol, Object
arg, int line){
            terminalsymbol.terminal.visit(this,null); return null;
      }
      public Object visitNonTerminalSymbol(NonTerminalSymbol nonterminalsymbol,
Object arg, int line){
            nonterminalsymbol.nonTerminal.visit(this,null); return null;
      }
      public Object visitNonTerminal(NonTerminal nonterminal, Object arg, int
line){
//          System.out.println("*NT*"+nonterminal+"*NT*");
            return null;
      }
      public Object visitTerminal(Terminal terminal, Object arg, int line){
//          System.out.println("*T*"+terminal+"*T*");
            return null;
```

```
	}

	public static void main(String[] args) throws Exception{
		Parser parser = new Parser();
		Grammar grammar = parser.parse();
		new Display(grammar);
		StarterSetChecker check = new StarterSetChecker();
		check.check(grammar);
		System.out.println("*****");
		StarterSetTable starterSetTable = check.findStarterSet();
		Iterator starterSetListIterator =
starterSetTable.getStarterSetEntryList().iterator();
		while (starterSetListIterator.hasNext()){
			StarterSetEntry starterSetEntry = (StarterSetEntry)
starterSetListIterator.next();
			System.out.println(starterSetEntry);
		}

		System.out.println("The program is terminated normally.");

	}

	public StarterSetTable findStarterSet(){
		//store starter ebnfsets after they are done
		StarterSetTable starterSetTable = new StarterSetTable();
		System.out.println("STARTSTARTSTARTSTARTSTARTSTARTSTARTSTARTSTARTSTART");
//		Iterator productionRuleListIterator = productionRuleList.iterator();
//		while (productionRuleListIterator.hasNext()){
//			ProductionRule productionRule = (ProductionRule)
productionRuleListIterator.next();
		for (int i=productionRuleList.size()-1;i>=0;i--){
			ProductionRule productionRule = (ProductionRule)
productionRuleList.get(i);

			StarterSet starterSet =
findStarterSet(productionRule.definition,starterSetTable);

			StarterSetEntry starterSetEntry = new
StarterSetEntry(productionRule.nonTerminal.spelling,starterSet);
			starterSetTable.addStarterSetEntry(starterSetEntry);

			//System.out.println(productionRule.nonTerminal.spelling+"==>
"+starterSet);
			System.out.println(starterSetEntry);

		}

	System.out.println("ENDENDENDENDENDENDENDENDENDENDENDENDENDENDENDENDENDEND
");
		System.out.println("starterSetTable:"+starterSetTable);
		int passcount = starterSetTable.getStarterSetEntryList().size();
		//remove remaining nonTerminals
		while (starterSetTable.containsUnresolvedStarterSetEntries() &&
passcount-->0){
			//Iterator starterSetListIterator =
starterSetTable.getStarterSetEntryList().iterator();
			//while (starterSetListIterator.hasNext()){
```

```
                  //    StarterSetEntry starterSetEntry = (StarterSetEntry)
starterSetListIterator.next();
                  for (int i=starterSetTable.getStarterSetEntryList().size()-
1;i>=0;i--){
                        StarterSetEntry starterSetEntry = (StarterSetEntry)
starterSetTable.getStarterSetEntryList().get(i);
                        //get keys for elements that are defined
                        Iterator starterSetMapKeysIterator =
starterSetTable.getNonTerminalNames().iterator();
                        while (starterSetMapKeysIterator.hasNext()){
                              String nonTerminalName = (String)
starterSetMapKeysIterator.next();
                              if
(starterSetEntry.containsNonTerminal(nonTerminalName)){
                                    System.out.println("*#*#*#Found
match:"+nonTerminalName);

      starterSetEntry.removeNonTerminal(nonTerminalName);

      starterSetEntry.getStarterSet().addAll(starterSetTable.getStarterSet(nonTe
rminalName));

                              }
                        }

                  }
            }

            if (passcount==-1)
                  System.out.println("Ended loop without finding all starter
sets.  All remaining non-terminals are tokens.");
            return starterSetTable;
      }

      public static StarterSet findStarterSet(Definition
definition,StarterSetTable starterSetTable){
            if (definition==null)
                  return null;
            StarterSet starterSet = new StarterSet();
            StarterSet childStarterSet =
findStarterSet(definition.definitionSequence,starterSetTable);
            starterSet.addAll(childStarterSet.getSymbolSet());
            if (childStarterSet.isCanBeEmpty()){
                  starterSet.setCanBeEmpty(true);
            }

            //Get alternatives
            DefinitionTail currentDefinitionTail = definition.definitionTail;
            while (currentDefinitionTail!=null){
                  childStarterSet =
findStarterSet(currentDefinitionTail.definitionSequence,starterSetTable);
                  starterSet.addAll(childStarterSet.getSymbolSet());
                  if (childStarterSet.isCanBeEmpty()){
                        starterSet.setCanBeEmpty(true);
                  }

                  currentDefinitionTail = currentDefinitionTail.definitionTail;
```

```
            }

            return starterSet;
      }

      /**
       * Get the StarterSet for a sequence A B C etc...
       * @param definitionSequence
       * @param starterSetTable
       * @return
       */
      public static StarterSet findStarterSet(DefinitionSequence
definitionSequence,StarterSetTable starterSetTable){
            StarterSet starterSet = new StarterSet();
            if (definitionSequence.definitionBody instanceof SymbolDefinition){
                  SymbolDefinition definitionBody = (SymbolDefinition)
definitionSequence.definitionBody;
                  Symbol symbol = definitionBody.symbol;
                  if (symbol instanceof TerminalSymbol){
                        TerminalSymbol terminalSymbol = (TerminalSymbol) symbol;
//                      starterSet.add(terminalSymbol.terminal.spelling);
                        starterSet.add(terminalSymbol);
                  }
                  if (symbol instanceof NonTerminalSymbol){
                        NonTerminalSymbol nonTerminalSymbol =
(NonTerminalSymbol) symbol;

                        StarterSet nonTerminalStarterSet =
starterSetTable.getStarterSet(nonTerminalSymbol.nonTerminal.spelling);
                        if (nonTerminalStarterSet!=null){
                              starterSet.addAll(nonTerminalStarterSet);

                        }
                        else{
//
      starterSet.add("**"+nonTerminalSymbol.nonTerminal.spelling+"**");
                              starterSet.add(nonTerminalSymbol);
                        }

                        //BLOCK ADDED 20040825
                        //if the nonterminalset can be empty get the other
values
                        if (nonTerminalStarterSet!=null &&
nonTerminalStarterSet.isCanBeEmpty()){
                              //add the rest to the starter set.
                              if (definitionSequence.definitionSequence!=null){
                                    StarterSet childStarterSet =
findStarterSet(definitionSequence.definitionSequence,starterSetTable);

      starterSet.addAll(childStarterSet.getSymbolSet());
                                    if (childStarterSet.isCanBeEmpty())
                                          starterSet.setCanBeEmpty(true);
                              } else {
                                    starterSet.setCanBeEmpty(true);

                              }
                        }
```

```
                }


            }
            if (definitionSequence.definitionBody instanceof
GroupingDefinition){
                GroupingDefinition definitionBody = (GroupingDefinition)
definitionSequence.definitionBody;
                //The starter set of a group is the starter set of the
definition in the group.

      starterSet=findStarterSet(definitionBody.definition,starterSetTable);

            }
            if (definitionSequence.definitionBody instanceof
ZeroOrOneDefinition){
                ZeroOrOneDefinition definitionBody = (ZeroOrOneDefinition)
definitionSequence.definitionBody;
                //The starter set of a group is the starter set of the
definition in the group + what comes after it.

      starterSet=findStarterSet(definitionBody.definition,starterSetTable);
                //add the rest to the starter set.
                if (definitionSequence.definitionSequence!=null){
                      StarterSet childStarterSet =
findStarterSet(definitionSequence.definitionSequence,starterSetTable);
                      starterSet.addAll(childStarterSet.getSymbolSet());
                      if (childStarterSet.isCanBeEmpty())
                            starterSet.setCanBeEmpty(true);
                } else {
                      starterSet.setCanBeEmpty(true);
                }

            }
            if (definitionSequence.definitionBody instanceof
ZeroOrMoreDefinition){
                ZeroOrMoreDefinition definitionBody = (ZeroOrMoreDefinition)
definitionSequence.definitionBody;
                //The starter set of a group is the starter set of the
definition in the group + what comes after it.

      starterSet=findStarterSet(definitionBody.definition,starterSetTable);
                //add the rest to the starter set.
                if (definitionSequence.definitionSequence!=null){
                      StarterSet childStarterSet =
findStarterSet(definitionSequence.definitionSequence,starterSetTable);
                      starterSet.addAll(childStarterSet.getSymbolSet());
                      if (childStarterSet.isCanBeEmpty())
                            starterSet.setCanBeEmpty(true);
                } else {
                      starterSet.setCanBeEmpty(true);
                }

            }
```

```
            return starterSet;
      }


      public static StarterSet findStarterSet(DefinitionBody
definitionBody,StarterSetTable starterSetTable){
            StarterSet starterSet = new StarterSet();
            if (definitionBody instanceof SymbolDefinition){
                  SymbolDefinition symbolDefinition = (SymbolDefinition)
definitionBody;
                  Symbol symbol = symbolDefinition.symbol;
                  if (symbol instanceof TerminalSymbol){
                        TerminalSymbol terminalSymbol = (TerminalSymbol) symbol;
                        starterSet.add(terminalSymbol);
                  }
                  if (symbol instanceof NonTerminalSymbol){
                        NonTerminalSymbol nonTerminalSymbol =
(NonTerminalSymbol) symbol;
                        StarterSet nonTerminalStarterSet =
starterSetTable.getStarterSet(nonTerminalSymbol.nonTerminal.spelling);
                        if (nonTerminalStarterSet!=null){
                              starterSet.addAll(nonTerminalStarterSet);

                        }
                        else{
                              starterSet.add(nonTerminalSymbol);
                        }
                  }

            }
            if (definitionBody instanceof GroupingDefinition){
                  GroupingDefinition groupingDefinition = (GroupingDefinition)
definitionBody;
                  //The starter set of a group is the starter set of the
definition in the group.

      starterSet=findStarterSet(groupingDefinition.definition,starterSetTable);

            }
            if (definitionBody instanceof ZeroOrOneDefinition){
                  ZeroOrOneDefinition zeroOrOneDefinition =
(ZeroOrOneDefinition) definitionBody;
                  //The starter set of a group is the starter set of the
definition in the group + what comes after it.

      starterSet=findStarterSet(zeroOrOneDefinition.definition,starterSetTable);
                  starterSet.setCanBeEmpty(true);

            }
            if (definitionBody instanceof ZeroOrMoreDefinition){
                  ZeroOrMoreDefinition zeroOrMoreDefinition =
(ZeroOrMoreDefinition) definitionBody;
                  //The starter set of a group is the starter set of the
definition in the group + what comes after it.
```

```
      starterSet=findStarterSet(zeroOrMoreDefinition.definition,starterSetTable)
;
                  starterSet.setCanBeEmpty(true);

            }

            return starterSet;
      }



}
```

ebnfsets.StarterSetEntry.java

```
/*
 * Created on Feb 9, 2004
 */
package ebnfsets;
import java.util.*;
import ebnfast.*;
/**
 * @author jfoure
 */
public class StarterSetEntry {
      private String nonTerminalName = null;
      private StarterSet starterSet = new StarterSet();
      public StarterSetEntry(String nonTerminalName, StarterSet starterSet){
            setNonTerminalName(nonTerminalName);
            setStarterSet(starterSet);
      }


      /**
       * @return
       */
      public String getNonTerminalName() {
            return nonTerminalName;
      }

      /**
       * @return
       */
      public StarterSet getStarterSet() {
            return starterSet;
      }

      /**
       * @param string
       */
      public void setNonTerminalName(String string) {
            nonTerminalName = string;
      }
```

```
	/**
	 * @param set
	 */
	public void setStarterSet(StarterSet set) {
		starterSet = set;
	}

	public boolean containsNonTerminal(){
		Iterator iterator = starterSet.getSymbolSet().iterator();
		while (iterator.hasNext()){
			Symbol symbol = (Symbol) iterator.next();
			if (symbol instanceof NonTerminalSymbol)
				return true;
		}

		return false;
	}

	public boolean containsNonTerminal(String nonTerminalName){
		Iterator iterator = starterSet.getSymbolSet().iterator();
		while (iterator.hasNext()){
			Symbol symbol = (Symbol) iterator.next();
			if (symbol instanceof NonTerminalSymbol)
			{
				if
(((NonTerminalSymbol)symbol).nonTerminal.spelling.equals(nonTerminalName))
					return true;
			}
		}

		return false;
	}

	public boolean removeNonTerminal(String nonTerminalName){
		NonTerminalSymbol nonTerminalSymbol = null;
		Iterator iterator = starterSet.getSymbolSet().iterator();
		while (iterator.hasNext()){
			Symbol symbol = (Symbol) iterator.next();
			if (symbol instanceof NonTerminalSymbol)
			{
				if
(((NonTerminalSymbol)symbol).nonTerminal.spelling.equals(nonTerminalName))
					nonTerminalSymbol = (NonTerminalSymbol)symbol;
			}
		}

		if (nonTerminalSymbol!=null)
			return starterSet.getSymbolSet().remove(nonTerminalSymbol);
		return false;
	}



	public String toString(){
		StringBuffer out = new StringBuffer();
		out.append("StarterSetEntry [ nonTerminalName:");
		out.append(nonTerminalName);
```

```
            out.append(" canBeEmpty:");
            out.append(starterSet.canBeEmpty);
            out.append(" containsNonTerminal:");
            out.append(containsNonTerminal());
            out.append(" symbolSet:");

            Iterator iterator = starterSet.getSymbolSet().iterator();
            while (iterator.hasNext()){
                  Symbol symbol = (Symbol) iterator.next();
                  if (symbol instanceof NonTerminalSymbol)

      out.append("**"+((NonTerminalSymbol)symbol).nonTerminal.spelling+"**");
                  if (symbol instanceof TerminalSymbol)
                        out.append(((TerminalSymbol)symbol).terminal.spelling);
                  if (iterator.hasNext())
                        out.append(", ");
            }
            out.append("]");
            return out.toString();
      }
}
```

ebnfsets.StarterSetTable.java

```
/*
 * Created on Feb 27, 2004
 */
package ebnfsets;
import ebnfast.*;
import common.*;

import java.util.*;
/**
 * @author jfoure
 */
public class StarterSetTable {
    protected List starterSetEntryList = new ArrayList();
    protected Map starterSetEntryMap = new HashMap();
    public void addStarterSetEntry(StarterSetEntry starterSetEntry) {
        starterSetEntryList.add(starterSetEntry);
        starterSetEntryMap.put(starterSetEntry.getNonTerminalName(),
starterSetEntry);
    }
    public List getStarterSetEntryList() {
        return starterSetEntryList;
    }
    public boolean containsUnresolvedStarterSetEntries() {
        Iterator starterSetEntryListIterator = starterSetEntryList.iterator();
        while (starterSetEntryListIterator.hasNext()) {
            StarterSetEntry starterSetEntry = (StarterSetEntry)
starterSetEntryListIterator.next();
            if (starterSetEntry.containsNonTerminal())
                return true;
        }
        return false;
```

```
    }
    public StarterSetEntry getStarterSetEntry(String nonTerminalName) {
        return (StarterSetEntry) starterSetEntryMap.get(nonTerminalName);
    }
    public StarterSet getStarterSet(String nonTerminalName) {
        if (getStarterSetEntry(nonTerminalName) != null)
            return getStarterSetEntry(nonTerminalName).getStarterSet();
        else
            return null;
    }
    public Set getNonTerminalNames() {
        return starterSetEntryMap.keySet();
    }
    public Set getUndefinedTerminalNames() {
        Set result = new HashSet();
        Iterator starterSetEntryListIterator = starterSetEntryList.iterator();
        while (starterSetEntryListIterator.hasNext()) {
            StarterSetEntry starterSetEntry = (StarterSetEntry)
starterSetEntryListIterator.next();
                  Iterator iterator =
starterSetEntry.getStarterSet().getSymbolSet().iterator();
                              while (iterator.hasNext()){
                                    Symbol symbol = (Symbol) iterator.next();
                                    if (symbol instanceof NonTerminalSymbol)

      result.add(((NonTerminalSymbol)symbol).nonTerminal.spelling);
                              }
        }

        return result;
    }

      public Set getUndefinedTerminalNames(GenericToken genericToken){
            Set result = new HashSet();
            Set undefinedTerminalNamesSet = getUndefinedTerminalNames();
            Iterator i = undefinedTerminalNamesSet.iterator();
            while (i.hasNext()){
                  String undefinedTerminalName = (String) i.next();
                  if
(genericToken.lookupConstantSpelling(undefinedTerminalName)==null)
                        result.add(undefinedTerminalName);
            }

            return result;
      }

}
```

ebnfsets.Test.java

```
/*
 * Created on Aug 24, 2004
 */
package ebnfsets;
import astast.Grammar;
```

```
import astast.Visitor;

import ebnfscanner.*;
import ebnfparser.*;
import ebnfast.*;
import ebnfsets.*;
import ebnf2ast.*;
import ast2parser.*;
import ebnftokengenerator.*;
import common.*;
import org.apache.log4j.*;
import java.util.*;

/**
 * @author jfoure
 */
public class Test {
      public static void main(String[] args) throws Exception{
            System.out.println("Creating parser");
            Parser parser = new Parser();
            System.out.println("Generating grammar");
            String g =
            "A ::= \"A\" B.\n"+
            "B ::= C | \"B\".\n"+
            "C ::= D \":=\" \"E\".\n"+
            "D ::= {\"F\"}.\n";

/*
            "Program ::= \"program\" ProgramName \";\" BlockBody \".\".\n"+
            "BlockBody ::= [ ConstantDefinitionPart ] [ TypeDefinitionPart ] [
VariableDefinitionPart ] { ProcedureDefinition }     CompoundStatement.\n"+
            "ConstantDefinitionPart ::= \"const\" ConstantDefinition {
ConstantDefinition }.\n"+
            "ConstantDefinition ::= Constant \"=\" Constant \";\".\n"+
            "TypeDefinitionPart ::= \"type\" TypeDefinition { TypeDefinition
}.\n"+
            "TypeDefinition ::= TypeName \"=\" NewType \";\".\n"+
            "NewType ::= NewArrayType | NewRecordType.\n"+
            "NewArrayType ::= \"array\" \"[\" IndexRange \"]\" \"of\"
TypeName.\n"+
            "IndexRange ::= Constant \"..\" Constant.\n"+
            "NewRecordType ::= \"record\" FieldList \"end\".\n"+
            "FieldList ::= RecordSection { \";\" RecordSection }.\n"+
            "RecordSection ::= FieldName { \",\" FieldName  } \":\"
TypeName.\n"+
            "VariableDefinitionPart ::= \"var\" VariableDefinitionPart {
VariableDefinitionPart }.\n"+
            "VariableDefinition ::= VariableGroup \";\".\n"+
            "VariableGroup ::= VariableName  { \",\" VariableName } \":\"
TypeName.\n"+
            "ProcedureDefinition ::= \"procedure\" ProcedureName ProcedureBlock
\";\".\n"+
            "ProcedureBlock ::= [ \"(\" FormalParameterList \")\" ] \";\"
BlockBody .\n"+
            "FormalParameterList ::= ParameterDefinition { \";\"
ParameterDefinition }.\n"+
            "ParameterDefinition ::= [\"var\"] VariableGroup.\n"+
```

```
            "Statement ::= [StatementBody] .\n"+
            "StatementBody ::= AssignmentOrProcedureStatement | IfStatement |
WhileStatement | CompoundStatement.\n"+
            "AssignmentOrProcedureStatement ::= Name
AssignmentOrProcedureStatementTail.\n"+
            "AssignmentOrProcedureStatementTail ::= AssignmentStatementTail |
ProcedureStatementTail.\n"+
            "AssignmentStatementTail ::= VariableAccessTail \":=\"
Expression.\n"+
            "ProcedureStatementTail ::= [ \"(\" ActualParameterList \")\" ].\n"+
            "ActualParameterList ::= ActualParameterList {\",\"
ActualParameter}.\n"+
            "ActualParameter ::= Expression.\n"+
            "IfStatement ::= \"if\" Expression \"then\" Statement [ \"else\"
Statement ].\n"+
            "WhileStatement ::= \"while\" Expression \"do\" Statement.\n"+
            "CompoundStatement ::= \"{\" Statement { \";\" Statement }
\"end\".\n"+
            "Expression ::= SimpleExpression [ RelationalOperator
SimpleExpression ].\n"+
            "RelationalOperator ::= \"<\" | \"=\" | \"<>\" | \">\".\n"+
            "SimpleExpression ::= [ SignOperator ] Term { AddingOperator Term
}.\n"+
            "SignOperator ::= \"+\" | \"-\".\n"+
            "AddingOperator ::= \"+\" | \"-\" | \"or\".\n"+
            "Term ::= Factor { MultiplyOperator Factor }.\n"+
            "MultiplyOperator ::= \"*\" | \"div\" | \"mod\" | \"and\".\n"+
            "Factor ::= Constant | VariableAccess | \"(\" Expression \")\" |
\"not\" Factor.\n"+
            "VariableAccess ::= VariableName VariableAccessTail.\n"+
            "VariableAccessTail ::= {Selector}.\n"+
            "Selector ::= IndexSelector | FieldSelector.\n"+
            "IndexSelector ::= \"[\" Expression \"]\".\n"+
            "FieldSelector ::= \".\" FieldName.\n"+
            "Constant ::= Numeral | ConstantName.\n"+
            "ProgramName ::= Identifier.\n"+
            "TypeName ::= Identifier.\n"+
            "FieldName ::= Identifier.\n"+
            "VariableName ::= Identifier.\n"+
            "ProcedureName ::= Identifier.\n"+
            "Name ::= Identifier.\n"+
            "ConstantName ::= ConstantIdentifier.\n";
*/

            ebnfast.Grammar grammar = parser.parse(g);
            System.out.println("Setting attribute");
            //new Display(grammar);
            StarterSetChecker check = new StarterSetChecker();
            check.check(grammar);
            System.out.println("*****");
            StarterSetTable starterSetTable = check.findStarterSet();

      System.out.println("starterSetTable:"+starterSetTable.getStarterSetEntryLi
st());
            System.out.println("****************************************");
            Iterator starterSetListIterator =
            starterSetTable.getStarterSetEntryList().iterator();
```

```
            while (starterSetListIterator.hasNext()) {
                  StarterSetEntry starterSetEntry =(StarterSetEntry)
starterSetListIterator.next();
            System.out.print(starterSetEntry.getNonTerminalName() + "
("+starterSetEntry.getStarterSet().isCanBeEmpty()+"):");
            Iterator iterator =
starterSetEntry.getStarterSet().getSymbolSet().iterator();
            while (iterator.hasNext()) {
                  Symbol symbol = (Symbol) iterator.next();
                  if (symbol instanceof NonTerminalSymbol)
                        System.out.print(((NonTerminalSymbol)
symbol).nonTerminal.spelling);
                  if (symbol instanceof TerminalSymbol)
                        System.out.print("\"" + ((TerminalSymbol)
symbol).terminal.spelling + "\"");
                  if (iterator.hasNext())
                        System.out.print(", ");
            }

                  System.out.println();
            }

      }
}
```

ebnftokengenerator.JoshScanner.java

```
package ebnftokengenerator;
import java.io.*;
import java.util.*;

import common.*;
import common.Error;
import common.GenericToken;
import common.SourceFile;
import common.TokenI;
import org.apache.log4j.*;

public class JoshScanner{
      public static Logger logger = Logger.getLogger("TokenGenerator");
      private List tokenRuleList = new ArrayList();

      private BufferedReader inFile;
      private char currentChar;
      //current tokenstring
      private StringBuffer currentTokenString;
      private static int line = 1;

      private List tokenList = new ArrayList();
      public JoshScanner(BufferedReader inFile,List tokenRuleList){
        this.inFile = inFile;
        this.tokenRuleList.addAll(tokenRuleList);
        logger.debug("tokenrulelist:"+this.tokenRuleList);
        try{
            int i = this.inFile.read();
```

```
            if(i == -1) //end of file
              currentChar = '\u0000';
            else
              currentChar = (char)i;
          }
          catch(IOException e){
                System.out.println(e);
          }
      }

      private void scanSeparator(){
            logger.debug("scanSeparator:"+currentChar);
          switch(currentChar){
            case '\n': case '\r':
              if(currentChar == '\n')
                  line++;
              discard();
          }
      }

      public TokenI scan(){
            logger.debug("scan:"+currentChar);
            currentTokenString = new StringBuffer("");
          while(currentChar == '\n' ||      currentChar == '\r')
            scanSeparator();

            if (tokenList.isEmpty()){
                  tokenList = scanToken();
            }

            TokenI token = (TokenI)tokenList.remove(0);
            return token;
      }


      public List scanToken(){
            logger.debug("scanToken:"+currentChar);
            logger.debug("scan1: "+currentTokenString+"/"+currentChar);
            currentTokenString = new StringBuffer("");
            GenericToken token = null;
          while(currentChar != '\n' && currentChar != '\r' && currentChar !=
'\u0000')
            takeIt();
            logger.debug(">["+currentTokenString+"]");
            while (currentTokenString.length()!=0){
                  if (currentTokenString.toString().equals("\u0000")){
                        tokenList.add(new GenericToken(TokenI.EOT,
currentTokenString.toString(), line));
                        return tokenList;
                  }

                  logger.debug(currentTokenString);
                  token = isMatch(currentTokenString.toString());
                  logger.debug("Match:"+token);
                  //System.out.println("Match:"+token);
                  currentTokenString =
currentTokenString.delete(0,token.spelling.length());
```

```
                tokenList.add(token);
            }

            if (token==null)
                  tokenList.add( new GenericToken(TokenI.EOT,
currentTokenString.toString(), line));
        //currentKind = scanToken();
        //return new Token(currentKind, currentSpelling.toString(), line);
        return tokenList;
      }

      private void discard(){
        try{
            int i = inFile.read();
            if(i == -1) //end of file
              currentChar = '\u0000';
            else
              currentChar = (char)i;
        }
        catch(IOException e){
              System.out.println(e);
        }
      }


      private void takeIt(){
            logger.debug("takeIt1: "+currentChar);
            currentTokenString.append(currentChar);
        try{
            int i = inFile.read();
            if(i == -1) //end of file
              currentChar = '\u0000';
            else
              currentChar = (char)i;
        }
        catch(IOException e){
              System.out.println(e);
        }
        logger.debug("takeIt2: "+currentChar);
      }

      private TokenI nextToken(){
            //currentTokenString
            return null;
      }

      /**
       * Loop through the rules and find the biggest match.
       * @param tokenString
       * @return
       */
      private GenericToken isMatch(String tokenString){
            logger.debug("isMatch:"+tokenString);
            if (tokenString.length()==0){
                  new Error("wrong token [" + currentChar+"]", line);
                  return new GenericToken(TokenI.EOT, tokenString, line);
            }
```

```
            for (int i=tokenRuleList.size()-1;i>-1;i--){
                  TokenRule tokenRule = (TokenRule) tokenRuleList.get(i);
                  logger.debug("Searching:
"+tokenRule.getRegularExpression()+"/"+tokenRule.getKind()+"/"+tokenString);
                  if (tokenString.matches(tokenRule.getRegularExpression())){
                        logger.debug("Found match:
"+tokenRule.getTokenName()+"/"+tokenRule.getRegularExpression()+"/"+tokenRule.ge
tKind()+"/"+tokenString);
                        return tokenRule.createToken(tokenString, line);
                  }
            }
            return isMatch(tokenString.substring(0,tokenString.length()-1));
      }


  public static void main(String[] args){
      Properties props = new Properties();
      //Set root logger level to DEBUG and its only appender to A1.
      props.put("log4j.rootLogger","DEBUG,A1");
      //A1 is set to be a ConsoleAppender.
      props.put("log4j.appender.A1","org.apache.log4j.ConsoleAppender");
      // A1 uses PatternLayout.
props.put("log4j.appender.A1.layout","org.apache.log4j.PatternLayout");
      props.put("log4j.appender.A1.layout.ConversionPattern","=*=>%-4r [%t] %-5p
%c %x - %m%n");
      PropertyConfigurator.configure(props);


    SourceFile sourceFile = new SourceFile();
    TokenI token;
    List tokenRuleList = new ArrayList();
      tokenRuleList.add(new TokenRule("Operator","\\x2B | - | * | / | // | \\ |
/\\ | \\/ | < | <= | > | >= | = | \\=",false,1,false));
      tokenRuleList.add(new TokenRule("string","[a-zA-
Z]+",false,2,false));//"\\w+",2));
      tokenRuleList.add(new TokenRule("IntegerLiteral","\\d+",false,3,false));
      tokenRuleList.add(new
TokenRule("CharacterLiteral","'\\w'",false,4,false));
      tokenRuleList.add(new TokenRule("Identifier","[a-zA-Z] ([a-zA-Z] |
\\d)*",false,5,false));
      tokenRuleList.add(new TokenRule("Comment","!(\\S? ?)*",false,6,true));
      tokenRuleList.add(new TokenRule("Seperator","\\x20",false,7,true));

    JoshScanner s = new JoshScanner(sourceFile.openFile(),tokenRuleList);
      System.out.println("tokenrulelist:"+tokenRuleList);

/*
      Comment: !\S*
      Operator:
      IntegerLiteral: \d+
      CharacterLiteral: '\w'
      Identifier: [a-zA-Z] ([a-zA-Z] | \d)*
*/

//http://java.sun.com/j2se/1.4.2/docs/api/java/util/regex/Pattern.html#sum
    do{
```

```
      token = s.scan();
      System.out.println("Line: " + token.getLine() + ", spelling = [" +
token.getSpelling() + "], " + "kind = " + token.getKind());
    }while(token.getKind() != TokenI.EOT);
  }
}
```

ebnftokengenerator.TestScanner.java

```
package ebnftokengenerator;
import java.io.*;
import java.util.*;

import common.*;
import common.Error;
import ebnftokengenerator.*;
import org.apache.log4j.*;

public class TestScanner{
      public static Logger logger = Logger.getLogger("TokenGenerator");
      private List tokenRuleList = new ArrayList();

      private BufferedReader inFile;
      private char currentChar;
      //current tokenstring
      private StringBuffer currentTokenString;
      private static int line = 1;

      private List tokenList = new ArrayList();
      public TestScanner(BufferedReader inFile,List tokenRuleList){
        this.inFile = inFile;
        this.tokenRuleList.addAll(tokenRuleList);
        logger.debug("tokenrulelist:"+this.tokenRuleList);
        try{
            int i = this.inFile.read();
            if(i == -1) //end of file
              currentChar = '\u0000';
            else
              currentChar = (char)i;
        }
        catch(IOException e){
              System.out.println(e);
        }
      }

      private void scanSeparator(){
            logger.debug("scanSeparator:"+currentChar);
        switch(currentChar){
            case '\n': case '\r':
              if(currentChar == '\n')
                  line++;
              discard();
        }
      }
```

```
      public TokenI scan(){
            logger.debug("scan:"+currentChar);
            currentTokenString = new StringBuffer("");
        while(currentChar == '\n' ||      currentChar == '\r')
            scanSeparator();

            if (tokenList.isEmpty()){
                  tokenList = scanToken();
            }

            TokenI token = (TokenI)tokenList.remove(0);
            return token;
      }


      public List scanToken(){
            logger.debug("scanToken:"+currentChar);
            logger.debug("scan1: "+currentTokenString+"/"+currentChar);
            currentTokenString = new StringBuffer("");
            GenericToken token = null;
        while(currentChar != '\n' && currentChar != '\r' && currentChar !=
'\u0000')
            takeIt();
            logger.debug(">["+currentTokenString+"]");
            while (currentTokenString.length()!=0){
                  if (currentTokenString.toString().equals("\u0000")){
                        tokenList.add(new GenericToken(TokenI.EOT,
currentTokenString.toString(), line));
                        return tokenList;
                  }

                  logger.debug(currentTokenString);
                  token = isMatch(currentTokenString.toString());
                  logger.debug("Match:"+token);
                  //System.out.println("Match:"+token);
                  currentTokenString =
currentTokenString.delete(0,token.spelling.length());
                  tokenList.add(token);
            }

            if (token==null)
                  tokenList.add( new GenericToken(TokenI.EOT,
currentTokenString.toString(), line));
        //currentKind = scanToken();
        //return new Token(currentKind, currentSpelling.toString(), line);
        return tokenList;
      }

      private void discard(){
        try{
            int i = inFile.read();
            if(i == -1) //end of file
              currentChar = '\u0000';
            else
              currentChar = (char)i;
        }
        catch(IOException e){
```

```
            System.out.println(e);
        }
    }


    private void takeIt(){
          logger.debug("takeIt1: "+currentChar);
          currentTokenString.append(currentChar);
      try{
          int i = inFile.read();
          if(i == -1) //end of file
            currentChar = '\u0000';
          else
            currentChar = (char)i;
      }
      catch(IOException e){
            System.out.println(e);
      }
      logger.debug("takeIt2: "+currentChar);
    }

    private TokenI nextToken(){
          //currentTokenString
          return null;
    }

    /**
     * Loop through the rules and find the biggest match.
     * @param tokenString
     * @return
     */
    private GenericToken isMatch(String tokenString){
          logger.debug("isMatch:"+tokenString);
          if (tokenString.length()==0){
                new Error("wrong token [" + currentChar+"]", line);
                return new GenericToken(TokenI.EOT, tokenString, line);
          }

          for (int i=tokenRuleList.size()-1;i>-1;i--){
                TokenRule tokenRule = (TokenRule) tokenRuleList.get(i);
                logger.debug("Searching:
"+tokenRule.getRegularExpression()+"/"+tokenRule.getKind()+"/"+tokenString);
                if (tokenString.matches(tokenRule.getRegularExpression())){
                      logger.debug("Found match:
"+tokenRule.getTokenName()+"/"+tokenRule.getRegularExpression()+"/"+tokenRule.ge
tKind()+"/"+tokenString);
                      return tokenRule.createToken(tokenString, line);
                }
          }
          return isMatch(tokenString.substring(0,tokenString.length()-1));
    }


  public static void main(String[] args){
      Properties props = new Properties();
      //Set root logger level to DEBUG and its only appender to A1.
      props.put("log4j.rootLogger","DEBUG,A1");
```

```
      //A1 is set to be a ConsoleAppender.
      props.put("log4j.appender.A1","org.apache.log4j.ConsoleAppender");
      // A1 uses PatternLayout.
props.put("log4j.appender.A1.layout","org.apache.log4j.PatternLayout");
      props.put("log4j.appender.A1.layout.ConversionPattern","=*=>%-4r [%t] %-5p
%c %x - %m%n");
      //PropertyConfigurator.configure(props);


    SourceFile sourceFile = new SourceFile();
    TokenI token;
    List tokenRuleList = new ArrayList();
      tokenRuleList.add(new TokenRule("SEPERATOR","\\x20|\\t",false,-2,true));
      tokenRuleList.add(new TokenRule("COMMENT","!(\\S? ?)*",false,-3,true));
      tokenRuleList.add(new TokenRule("IDENTIFIER","[a-zA-Z]([a-zA-
Z]|\\d)*",false,0,false));
      tokenRuleList.add(new TokenRule("INTEGERLITERAL","\\d+",false,1,false));
      tokenRuleList.add(new
TokenRule("CHARACTERLITERAL","'\\w'",false,2,false));
      tokenRuleList.add(new TokenRule("NOTUSED","\\a",false,3,false));
      tokenRuleList.add(new TokenRule("OPERATOR","\\x2B|-
|\\x2A|/|//|\\|/\\|\\/|<|<=|>|>=|=|\\=",false,4,false));
      tokenRuleList.add(new TokenRule("ARRAY","array",true,5,false));
      tokenRuleList.add(new TokenRule("BEGIN","begin",true,6,false));
      tokenRuleList.add(new TokenRule("CONST","const",true,7,false));
      tokenRuleList.add(new TokenRule("DO","do",true,8,false));
      tokenRuleList.add(new TokenRule("ELSE","else",true,9,false));
      tokenRuleList.add(new TokenRule("END","end",true,10,false));
      tokenRuleList.add(new TokenRule("IF","if",true,11,false));
      tokenRuleList.add(new TokenRule("IN","in",true,12,false));
      tokenRuleList.add(new TokenRule("LET","let",true,13,false));
      tokenRuleList.add(new TokenRule("OF","of",true,14,false));
      tokenRuleList.add(new TokenRule("PROC","proc",true,15,false));
      tokenRuleList.add(new TokenRule("THEN","then",true,16,false));
      tokenRuleList.add(new TokenRule("VAR","var",true,17,false));
      tokenRuleList.add(new TokenRule("WHILE","while",true,18,false));
      tokenRuleList.add(new TokenRule("COLON",":",true,19,false));
      tokenRuleList.add(new TokenRule("SEMICOLON",";",true,20,false));
      tokenRuleList.add(new TokenRule("COMMA",",",true,21,false));
      tokenRuleList.add(new TokenRule("BECOMES",":=",true,22,false));
      tokenRuleList.add(new TokenRule("NOT","~",true,23,false));
      tokenRuleList.add(new TokenRule("LPAREN","\\x28",true,24,false));
      tokenRuleList.add(new TokenRule("RPAREN","\\x29",true,25,false));
      tokenRuleList.add(new TokenRule("LBRACKET","\\x5B",true,26,false));
      tokenRuleList.add(new TokenRule("RBRACKET","\\x5D",true,27,false));
    TestScanner s = new TestScanner(sourceFile.openFile(),tokenRuleList);
      System.out.println("tokenrulelist:"+tokenRuleList);
//http://java.sun.com/j2se/1.4.2/docs/api/java/util/regex/Pattern.html#sum
    do{
      token = s.scan();
      if (token.getKind()>=-1)
            System.out.println("Line: " + token.getLine() + ", spelling = [" +
token.getSpelling() + "], " + "kind = " + token.getKind());
    }while(token.getKind() != TokenI.EOT);
  }
}
```

ebnftokengenerator.TestToken.java

```
package ebnftokengenerator;
import common.*;
import common.Error;
import ebnftokengenerator.*;
import java.util.*;

public class TestToken implements TokenI{
public int kind;
public String spelling;
public int line;
public String toString(){
      return "TestToken line: "+line+" kind: "+kind+" spelling: "+spelling;
}

public TestToken(){
}

public TestToken(byte kind, String spelling, int line){
this.kind = kind;
this.spelling = spelling;
this.line = line;
}

public String lookupConstantSpelling(String spelling){
      String result = (String) constantSpellingMap.get(spelling);
      return result;
}
public String lookupVariableSpelling(String spelling){
      return (String)variableSpellingMap.get(spelling);
}
public final static byte
      EOT = -1,
      SEPERATOR = 1,          //\x20|\t
      COMMENT = 2,            //!(\S? ?)*
      IDENTIFIER = 3,         //[a-zA-Z]([a-zA-Z]|\d)*
      INTEGERLITERAL = 4,           //\d+
      CHARACTERLITERAL = 5,         //'\w'
      NOTUSED = 6,            //\a
      OPERATOR = 7,           //\x2B|-|\x2A|/|//|\|/\|\/|<|<=|>|>=|=|\=
      ARRAY = 8,        //array
      BEGIN = 9,        //begin
      CONST = 10,       //const
      DO = 11,          //do
      ELSE = 12,        //else
      END = 13,         //end
      IF = 14,          //if
      IN = 15,          //in
      LET = 16,         //let
      OF = 17,          //of
      PROC = 18,        //proc
      THEN = 19,        //then
      VAR = 20,         //var
      WHILE = 21,       //while
```

```
      COLON = 22,        //:
      SEMICOLON = 23,          //;
      COMMA = 24,        //,
      BECOMES = 25,            //:=
      NOT = 26,          //~
      LPAREN = 27,             //\x28
      RPAREN = 28,             //\x29
      LBRACKET = 29,           //\x5B
      RBRACKET = 30;           //\x5D
private final static Map constantSpellingMap = new HashMap();
static {
      constantSpellingMap.put("\\x20|\\t","SEPERATOR");
      constantSpellingMap.put("!(\\S? ?)*","COMMENT");
      constantSpellingMap.put("[a-zA-Z]([a-zA-Z]|\\d)*","IDENTIFIER");
      constantSpellingMap.put("\\d+","INTEGERLITERAL");
      constantSpellingMap.put("'\\w'","CHARACTERLITERAL");
      constantSpellingMap.put("\\a","NOTUSED");
      constantSpellingMap.put("\\x2B|-
|\\x2A|/|//|\\|/\\|\\/|<|<=|>|>=|=|\\=","OPERATOR");
      constantSpellingMap.put("array","ARRAY");
      constantSpellingMap.put("begin","BEGIN");
      constantSpellingMap.put("const","CONST");
      constantSpellingMap.put("do","DO");
      constantSpellingMap.put("else","ELSE");
      constantSpellingMap.put("end","END");
      constantSpellingMap.put("if","IF");
      constantSpellingMap.put("in","IN");
      constantSpellingMap.put("let","LET");
      constantSpellingMap.put("of","OF");
      constantSpellingMap.put("proc","PROC");
      constantSpellingMap.put("then","THEN");
      constantSpellingMap.put("var","VAR");
      constantSpellingMap.put("while","WHILE");
      constantSpellingMap.put(":","COLON");
      constantSpellingMap.put(";","SEMICOLON");
      constantSpellingMap.put(",","COMMA");
      constantSpellingMap.put(":=","BECOMES");
      constantSpellingMap.put("~","NOT");
      constantSpellingMap.put("\\x28","LPAREN");
      constantSpellingMap.put("\\x29","RPAREN");
      constantSpellingMap.put("\\x5B","LBRACKET");
      constantSpellingMap.put("\\x5D","RBRACKET");
};

private final static Map variableSpellingMap = new HashMap();
static {
      variableSpellingMap.put("SEPERATOR","\\x20|\\t");
      variableSpellingMap.put("COMMENT","!(\\S? ?)*");
      variableSpellingMap.put("IDENTIFIER","[a-zA-Z]([a-zA-Z]|\\d)*");
      variableSpellingMap.put("INTEGERLITERAL","\\d+");
      variableSpellingMap.put("CHARACTERLITERAL","'\\w'");
      variableSpellingMap.put("NOTUSED","\\a");
      variableSpellingMap.put("OPERATOR","\\x2B|-
|\\x2A|/|//|\\|/\\|\\/|<|<=|>|>=|=|\\=");
      variableSpellingMap.put("ARRAY","array");
      variableSpellingMap.put("BEGIN","begin");
      variableSpellingMap.put("CONST","const");
```

```
	variableSpellingMap.put("DO","do");
	variableSpellingMap.put("ELSE","else");
	variableSpellingMap.put("END","end");
	variableSpellingMap.put("IF","if");
	variableSpellingMap.put("IN","in");
	variableSpellingMap.put("LET","let");
	variableSpellingMap.put("OF","of");
	variableSpellingMap.put("PROC","proc");
	variableSpellingMap.put("THEN","then");
	variableSpellingMap.put("VAR","var");
	variableSpellingMap.put("WHILE","while");
	variableSpellingMap.put("COLON",":");
	variableSpellingMap.put("SEMICOLON",";");
	variableSpellingMap.put("COMMA",",");
	variableSpellingMap.put("BECOMES",":=");
	variableSpellingMap.put("NOT","~");
	variableSpellingMap.put("LPAREN","\\x28");
	variableSpellingMap.put("RPAREN","\\x29");
	variableSpellingMap.put("LBRACKET","\\x5B");
	variableSpellingMap.put("RBRACKET","\\x5D");
};

public int getKind() {
	return kind;
}

public int getLine() {
	return line;
}

public String getSpelling() {
	return spelling;
}

}
```

ebnftokengenerator.TokenDefinition.java

```
/*
 * Created on Aug 2, 2004
 */
package ebnftokengenerator;

/**
 * @author jfoure
 */
public abstract class TokenDefinition {
	public TokenDefinition(){
	}

	public static TokenDefinition createDefinition(String definition){
		if (definition.startsWith("\"") && definition.endsWith("\"")){
			return new
TokenLiteralDefinition(definition.substring(1,definition.length()-1));
		} else
```

```
                  return new TokenVariableDefinition(definition);

      }

}
```

ebnftokengenerator.TokenEntry.java

```
/*
 * Created on Aug 2, 2004
 */
package ebnftokengenerator;

/**
 * @author jfoure
 */
public class TokenEntry {
      String variableName = null;
      String initialDefinition = null;
      TokenDefinition tokenDefinition = null;
      boolean ignored = false;
      boolean reservedWord = true;

      public TokenEntry(String variableName, String initialDefinition,
TokenDefinition tokenDefinition, boolean ignored){
            this.variableName = variableName;
            this.initialDefinition = initialDefinition;
            this.tokenDefinition = tokenDefinition;
            this.ignored = ignored;
      }

      public String toString(){
            return
"TokenEntry["+variableName+","+tokenDefinition+","+ignored+"]";
      }

      /**
       * @return
       */
      public boolean isIgnored() {
            return ignored;
      }

      /**
       * @return
       */
      public TokenDefinition getTokenDefinition() {
            return tokenDefinition;
      }

      /**
       * @return
       */
      public String getVariableName() {
            return variableName;
```

```
	}

	/**
	 * @param b
	 */
	public void setIgnored(boolean b) {
		ignored = b;
	}

	/**
	 * @param definition
	 */
	public void setTokenDefinition(TokenDefinition definition) {
		tokenDefinition = definition;
	}

	/**
	 * @param string
	 */
	public void setVariableName(String string) {
		variableName = string;
	}

	/**
	 * @return
	 */
	public boolean isReservedWord() {
		return reservedWord;
	}

	/**
	 * @param b
	 */
	public void setReservedWord(boolean b) {
		reservedWord = b;
	}

	/**
	 * @return
	 */
	public String getInitialDefinition() {
		return initialDefinition;
	}

	/**
	 * @param string
	 */
	public void setInitialDefinition(String string) {
		initialDefinition = string;
	}

}
```

ebnftokengenerator.TokenGenerator.java

```
/*
 * Created on Aug 1, 2004
 */
package ebnftokengenerator;
import java.io.*;
import java.util.*;

import common.*;
import org.apache.log4j.*;
/**
 * @author jfoure
 */
public class TokenGenerator {
	public static Logger logger = Logger.getLogger("TokenGenerator");

	public static String encodeJavaString(String s){
		StringBuffer out = new StringBuffer();
		/*
		result = s.replaceAll("\\x5C","AAA");  //   \\ --> \\\\
		result = s.replaceAll("\\\\","CCC");  //   \\ --> \\\\
		result = s.replaceAll("\\x22","BBB");
		*/
		int chr = -1;
		for (int i=0;i<s.length();i++){
			chr = s.charAt(i);
			switch (chr){
				case '\\':out.append("\\\\");break;
				case '\"':out.append("\\\"");break;
				default:out.append((char)chr);
			}
		}

		return out.toString();
	}

	public static GenericToken load(String settings) throws Exception{
		GenericToken genericToken = new GenericToken();
		List tokenList = TokenGenerator.parseTokenRuleList(new
BufferedReader(new StringReader(settings)));
		Iterator tokenListIterator = tokenList.iterator();
		for (int i=1;tokenListIterator.hasNext();i++){
			TokenEntry tokenEntry = (TokenEntry)tokenListIterator.next();
			String constantSpelling = tokenEntry.getVariableName();
			String literal =
((TokenLiteralDefinition)tokenEntry.getTokenDefinition()).getRule();

	GenericToken.constantSpellingMap.put(literal,constantSpelling);

	GenericToken.variableSpellingMap.put(literal,constantSpelling.toLowerCase(
));

		}

		GenericToken.tokenList = tokenList;
		return genericToken;
	}
```

```
	public static List parseTokenRuleList(BufferedReader in) throws Exception{

		String line = null;
		List tokenList = new ArrayList();
		Map tokenVariableMap = new HashMap();
		boolean isIgnoredTokenEntry = false; boolean isTokenEntry = false;
		while ( (line=in.readLine())!=null){
			System.out.println(line);
			if (line.length()==0)
				continue;
			if (line.equals("#IGNORED TOKENS")){
				isIgnoredTokenEntry = true;
				isTokenEntry = false;
			}
			else if (line.startsWith("#TOKENS")){
				isIgnoredTokenEntry = false;
				isTokenEntry = true;
			}
			else if (line.startsWith("#VARIABLES")){
				isIgnoredTokenEntry = false;
				isTokenEntry = false;
			}
			else if (line.startsWith("#")){
				//do nothing.  this is a comment
			}
			else{
				if (isIgnoredTokenEntry){
					TokenEntry tokenEntry = new
TokenEntry(line.substring(0,line.indexOf(":")),line.substring(line.indexOf(":")+
1),TokenDefinition.createDefinition(line.substring(line.indexOf(":")+1)),true);
					tokenList.add(tokenEntry);
					System.out.println(tokenEntry);
				} else if (isTokenEntry){
					TokenEntry tokenEntry = new
TokenEntry(line.substring(0,line.indexOf(":")),line.substring(line.indexOf(":")+
1),TokenDefinition.createDefinition(line.substring(line.indexOf(":")+1)),false);
					tokenList.add(tokenEntry);
					System.out.println(tokenEntry);
				} else {
					String currentTokenVariable =
line.substring(0,line.indexOf(":"));
					TokenVariableEntry tokenVariableEntry = new
TokenVariableEntry(currentTokenVariable,line.substring(line.indexOf(":")+1));

	tokenVariableMap.put(currentTokenVariable,tokenVariableEntry);
					System.out.println(tokenVariableMap);
				}

			}

		}

		logger.debug("Tokens:"+tokenList);
		logger.debug("Token Variables:"+tokenVariableMap);
		//remove variable definitions and set reservedword to false
		Iterator tokenListIterator = tokenList.iterator();
```

```
            while (tokenListIterator.hasNext()){
                  TokenEntry tokenEntry = (TokenEntry)tokenListIterator.next();
                  if (tokenEntry.tokenDefinition instanceof
TokenVariableDefinition){
                        TokenVariableDefinition tokenVariableDefinition =
(TokenVariableDefinition) tokenEntry.tokenDefinition;
                        tokenEntry.tokenDefinition = new TokenLiteralDefinition(
((TokenVariableEntry)
tokenVariableMap.get(tokenVariableDefinition.variableName)).rule);
                        tokenEntry.setReservedWord(false);
                  }

            }

            logger.debug("Tokens:"+tokenList);
            logger.debug("Token Variables:"+tokenVariableMap);
            return tokenList;
      }

      public static GeneratedFile generateToken(String tokenFilename, String
packageName, List tokenList){
            StringBuffer out = new StringBuffer(); out.append("package
"+packageName+";\n\n"); out.append("import common.*;\n"); out.append("import
java.util.*;\n\n");
            out.append("public class ");
            out.append(tokenFilename);
            out.append(" implements TokenI{\n\n");

            out.append("public int kind;\n");
            out.append("public String spelling;\n");
            out.append("public int line;\n");
            out.append("public String toString(){\n");
            out.append("\treturn \""+tokenFilename+" line: \"+line+\" kind:
\"+kind+\" spelling: \"+spelling;\n");
            out.append("}\n\n");
            out.append("public "+tokenFilename+"(){\n"); out.append("}\n\n");
            out.append("public "+tokenFilename+"(int kind, String spelling, int
line){\n");
            out.append("this.kind = kind;\n");
            out.append("this.spelling = spelling;\n");
            out.append("this.line = line;\n");
            out.append("}\n\n");

            out.append("public String lookupConstantSpelling(String
spelling){\n");
            out.append("\tString result = (String)
constantSpellingMap.get(spelling);\n");
            out.append("\treturn result;\n");
            out.append("}\n");

            out.append("public String lookupVariableSpelling(String
spelling){\n");
            out.append("\treturn (String)variableSpellingMap.get(spelling);\n");
            out.append("}\n");
            out.append("public final static byte\n"); out.append("\tEOT = -
1,\n");
            Iterator tokenListIterator = tokenList.iterator();
```

```
            int tokenCount = 0;
            int ignoredTokenCount = -2;
            tokenListIterator = tokenList.iterator();
            for (int i=1;tokenListIterator.hasNext();i++){
                  TokenEntry tokenEntry = (TokenEntry)tokenListIterator.next();
                  out.append("\t"+tokenEntry.variableName+" = ");
                  if (tokenEntry.ignored)
                        out.append(ignoredTokenCount--);
                  else
                        out.append(tokenCount++);
                  if (tokenListIterator.hasNext())

      out.append(",\t\t//"+((TokenLiteralDefinition)tokenEntry.tokenDefinition).
rule+"\n");
                  else

      out.append(";\t\t//"+((TokenLiteralDefinition)tokenEntry.tokenDefinition).
rule+"\n");

            }
/*
            Iterator tokenListIterator = tokenList.iterator();
            for (int i=1;tokenListIterator.hasNext();i++){
                  TokenEntry tokenEntry = (TokenEntry)tokenListIterator.next();
                  out.append("\t"+tokenEntry.variableName+" = "+i);
                  if (tokenListIterator.hasNext())

      out.append(",\t\t//"+((TokenLiteralDefinition)tokenEntry.tokenDefinition).
rule+"\n");
                  else

      out.append(";\t\t//"+((TokenLiteralDefinition)tokenEntry.tokenDefinition).
rule+"\n");
            }
*/



/*
            private final static String[] spellings = {
              "<identifier>", "<intliteral>", "<charliteral>", "<operator>",
              "array", "begin", "const", "do", "else", "end", "if", "in", "let",
"of",
              "proc", "skip", "then", "var", "while"
            };
*/

            out.append("private final static Map constantSpellingMap = new
HashMap();\n");
            out.append("static {\n");
            tokenListIterator = tokenList.iterator();
            for (int i=1;tokenListIterator.hasNext();i++){
                  TokenEntry tokenEntry = (TokenEntry)tokenListIterator.next();

      out.append("\tconstantSpellingMap.put(\""+encodeJavaString(((TokenLiteralD
efinition)tokenEntry.tokenDefinition).rule)+"\",\""+tokenEntry.variableName+"\")
;\n");
```

```
            }
            out.append("};\n\n");
            out.append("private final static Map variableSpellingMap = new
HashMap();\n");
            out.append("static {\n");
            tokenListIterator = tokenList.iterator();
            for (int i=1;tokenListIterator.hasNext();i++){
                  TokenEntry tokenEntry = (TokenEntry)tokenListIterator.next();

      out.append("\tvariableSpellingMap.put(\""+tokenEntry.variableName+"\",\""+
encodeJavaString(((TokenLiteralDefinition)tokenEntry.tokenDefinition).rule)+"\")
;\n");
            }
            out.append("};\n\n");

            out.append("public int getKind() {\n");
            out.append("\treturn kind;\n");
            out.append("}\n\n");
            out.append("public int getLine() {\n");
            out.append("\treturn line;\n");
            out.append("}\n\n");
            out.append("public String getSpelling() {\n");
            out.append("\treturn spelling;\n");
            out.append("}\n\n");
            out.append("}");
            return new
GeneratedFile(tokenFilename+".java",packageName,out.toString());
      }

      public static GeneratedFile generateScanner(String scannerFilename, String
packageName, String sourceFilename, List tokenList){
            StringBuffer out = new StringBuffer();
            out.append("package "+packageName+";\n");
            out.append("\n");
            out.append("import java.io.*;\n");
            out.append("import java.util.*;\n");
            out.append("\n");
            out.append("import common.TokenI;\n");
            out.append("import common.Error;\n");
            out.append("import common.TokenRule;\n");
//          out.append("import org.apache.log4j.*;\n");
            out.append("\n");
            out.append("public class "+scannerFilename+"{\n");
//          out.append("\tpublic static Logger logger =
Logger.getLogger(\"Scanner\");\n");
            out.append("\t\n");
            out.append("\tprivate List tokenRuleList = new ArrayList();\t\n");
            out.append("\t\n");
            out.append("\t\n");
            out.append("\tprivate BufferedReader inFile;\n");
            out.append("\tprivate char currentChar;\n");
            out.append("\t//current tokenstring\n");
            out.append("\tprivate StringBuffer currentTokenString;\n");
            out.append("\tprivate static int line = 1;\n");
            out.append("\t\n");
            out.append("\tprivate List tokenList = new ArrayList();\n");
            out.append("\t\n");
```

```
            out.append("\tpublic "+scannerFilename+"(BufferedReader
inFile){\n");
            out.append("    List tokenRuleList = new ArrayList();\n");
            Iterator tokenListIterator = tokenList.iterator();
            int tokenCount = 0;
            int ignoredTokenCount = -2;
            tokenListIterator = tokenList.iterator();
            for (int i=1;tokenListIterator.hasNext();i++){
                  TokenEntry tokenEntry = (TokenEntry)tokenListIterator.next();
                  if (tokenEntry.ignored)
                        out.append("\ttokenRuleList.add(new
TokenRule(\""+tokenEntry.variableName+"\",\""+encodeJavaString(((TokenLiteralDef
inition)tokenEntry.tokenDefinition).rule)+"\","+tokenEntry.isReservedWord()+","+
(ignoredTokenCount--)+","+tokenEntry.ignored+"));\n");
                  else
                        out.append("\ttokenRuleList.add(new
TokenRule(\""+tokenEntry.variableName+"\",\""+encodeJavaString(((TokenLiteralDef
inition)tokenEntry.tokenDefinition).rule)+"\","+tokenEntry.isReservedWord()+","+
(tokenCount++)+","+tokenEntry.ignored+"));\n");
            }
            out.append("\t  this.inFile = inFile;\n");
            out.append("\t  this.tokenRuleList.addAll(tokenRuleList);\n");
//          out.append("\t
logger.debug(\"tokenrulelist:\"+this.tokenRuleList);\n");
            out.append("\t  try{\n");
            out.append("\t\tint i = this.inFile.read();\n");
            out.append("\t\tif(i == -1) //end of file\n");
            out.append("\t\t  currentChar = '\\u0000';\n");
            out.append("\t\telse\n");
            out.append("\t\t  currentChar = (char)i;\n");
            out.append("\t  }\n");
            out.append("\t  catch(IOException e){\n");
            out.append("\t\t  System.out.println(e);\n");
            out.append("\t  }\n");
            out.append("\t}\n");
            out.append("\n");
            out.append("\tprivate void scanSeparator(){\n");
//          out.append("\t\tlogger.debug(\"scanSeparator:\"+currentChar);\n");
            out.append("\t  switch(currentChar){\n");
            out.append("\t\tcase '\\n': case '\\r':\n");
            out.append("\t\t  if(currentChar == '\\n')\n");
            out.append("\t\t\tline++;\n");
            out.append("\t\t  discard();\n");
            out.append("\t  }\n");
            out.append("\t}\n");
            out.append("\n");
            out.append("\tpublic TokenI scanAll(){\n");
//          out.append("\t\tlogger.debug(\"scan:\"+currentChar);\n");
            out.append("\t\tcurrentTokenString = new StringBuffer(\"\");\n");
            out.append("\t  while(currentChar == '\\n' ||\tcurrentChar ==
'\\r')\n");
            out.append("\t\tscanSeparator();\n");
            out.append("\t\t\n");
            out.append("\t\tif (tokenList.isEmpty()){\n");
            out.append("\t\t\ttokenList = scanToken();\n");
            out.append("\t\t}\n");
            out.append("\n");
```

```
            out.append("\t\tTokenI token = (TokenI)tokenList.remove(0);\n");
            out.append("\t\treturn token;\n");
            out.append("\t}\n");
            out.append("\n");
            out.append("\tpublic TokenI scan(){\n");
            out.append("\t\tTokenI token = scanAll();\n");
            out.append("\t\twhile (token.getKind()<-1)\n");
            out.append("\t\ttoken = scanAll();\n");
            out.append("\n");
            out.append("\treturn token; \n");
            out.append("\t}\n");
            out.append("\n");
            out.append("\tpublic List scanToken(){\n");
//          out.append("\t\tlogger.debug(\"scanToken:\"+currentChar);\n");
//          out.append("\t\tlogger.debug(\"scan1:
\"+currentTokenString+\"/\"+currentChar);\n");
            out.append("\t\tcurrentTokenString = new StringBuffer(\"\");\n");
            out.append("\t\tToken token = null;\n");
            out.append("\t  while(currentChar != '\\n' && currentChar != '\\r'
&& currentChar != '\\u0000')\n");
            out.append("\t\ttakeIt();\n");
            out.append("\t\t\n");
//          out.append("\t\tlogger.debug(\">[\"+currentTokenString+\"]\");\n");
            out.append("\t\twhile (currentTokenString.length()!=0){\n");
            out.append("\t\t\tif
(currentTokenString.toString().equals(\"\\u0000\")){\n");
            out.append("\t\t\t\ttokenList.add(new Token(TokenI.EOT,
currentTokenString.toString(), line));\n");
            out.append("\t\t\t\treturn tokenList;\n");
            out.append("\t\t\t}\n");
            out.append("\n");
//          out.append("\t\t\tlogger.debug(currentTokenString);\n");
            out.append("\t\t\ttoken =
isMatch(currentTokenString.toString());\n");
//          out.append("\t\t\tlogger.debug(\"Match:\"+token);\n");
            out.append("\t\t\t//System.out.println(\"Match:\"+token);\n");
            out.append("\t\t\tcurrentTokenString =
currentTokenString.delete(0,token.spelling.length());\n");
            out.append("\t\t\t\n");
            out.append("\t\t\ttokenList.add(token);\n");
            out.append("\t\t}\n");
            out.append("\n");
            out.append("\t\tif (token==null)\n");
            out.append("\t\t\ttokenList.add( new Token(TokenI.EOT,
currentTokenString.toString(), line));\n");
            out.append("\t  //currentKind = scanToken();\n");
            out.append("\t  //return new Token(currentKind,
currentSpelling.toString(), line);\n");
            out.append("\t  return tokenList;\n");
            out.append("\t}\n");
            out.append("\n");
            out.append("\tprivate void discard(){\n");
            out.append("\t  try{\n");
            out.append("\t\tint i = inFile.read();\n");
            out.append("\t\tif(i == -1) //end of file\n");
            out.append("\t\t  currentChar = '\\u0000';\n");
            out.append("\t\telse\n");
```

```
            out.append("\t\t  currentChar = (char)i;\n");
            out.append("\t  }\n");
            out.append("\t  catch(IOException e){\n");
            out.append("\t\t  System.out.println(e);\n");
            out.append("\t  }\n");
            out.append("\t}\n");
            out.append("\n");
            out.append("\n");
            out.append("\tprivate void takeIt(){\n");
//          out.append("\t\tlogger.debug(\"takeIt1: \"+currentChar);\n");
            out.append("\t\tcurrentTokenString.append(currentChar);\n");
            out.append("\t  try{\n");
            out.append("\t\tint i = inFile.read();\n");
            out.append("\t\tif(i == -1) //end of file\n");
            out.append("\t\t  currentChar = '\\u0000';\n");
            out.append("\t\telse\n");
            out.append("\t\t  currentChar = (char)i;\n");
            out.append("\t  }\n");
            out.append("\t  catch(IOException e){\n");
            out.append("\t\t  System.out.println(e);\n");
            out.append("\t  }\n");
//          out.append("\t  logger.debug(\"takeIt2: \"+currentChar);\n");
            out.append("\t}\n");
            out.append("\t\n");
            out.append("\tprivate TokenI nextToken(){\n");
            out.append("\t\t//currentTokenString\n");
            out.append("\t\treturn null;\n");
            out.append("\t}\n");
            out.append("\n");
            out.append("\t/**\n");
            out.append("\t * Loop through the rules and find the biggest
match.\n");
            out.append("\t * @param tokenString\n");
            out.append("\t * @return\n");
            out.append("\t */\t\n");
            out.append("\tprivate Token isMatch(String tokenString){\n");
//          out.append("\t\tlogger.debug(\"isMatch:\"+tokenString);\n");
            out.append("\t\tif (tokenString.length()==0){\n");
            out.append("\t\t\tnew Error(\"wrong token [\" + tokenString+\"]\",
line);\n");
            out.append("\t\t\treturn new Token(TokenI.EOT, tokenString,
line);\n");
            out.append("\t\t}\n");
            out.append("\n");
            out.append("\t\tfor (int i=tokenRuleList.size()-1;i>-1;i--){\n");
            out.append("\t\t\tTokenRule tokenRule = (TokenRule)
tokenRuleList.get(i);\n");
//          out.append("\t\t\tlogger.debug(\"Searching:
\"+tokenRule.getRegularExpression()+\"/\"+tokenRule.getKind()+\"/\"+tokenString)
;\n");
            out.append("\t\t\tif ( (tokenRule.isReservedWord() &&
tokenString.equals(tokenRule.getRegularExpression())) ||
(!tokenRule.isReservedWord() &&
tokenString.matches(tokenRule.getRegularExpression())) ){\n");
//          out.append("\t\t\t\tlogger.debug(\"Found match:
\"+tokenRule.getTokenName()+\"/\"+tokenRule.getRegularExpression()+\"/\"+tokenRu
le.getKind()+\"/\"+tokenString);\n");
```

```
            out.append("\t\t\t\treturn new Token(tokenRule.getKind(),
tokenString, line);\n");
            out.append("\t\t\t}\n");
            out.append("\t\t}\n");
            out.append("\t\treturn
isMatch(tokenString.substring(0,tokenString.length()-1));\n");
            out.append("\t}\t\n");
            out.append("\n");
            out.append("\n");
            out.append("  public static void main(String[] args){\n");
//          out.append("\tProperties props = new Properties();\n");
//          out.append("\t//Set root logger level to DEBUG and its only appender
to A1.\n");
//          out.append("\tprops.put(\"log4j.rootLogger\",\"DEBUG,A1\");\n");
//          out.append("\t//A1 is set to be a ConsoleAppender.\n");
//
      out.append("\tprops.put(\"log4j.appender.A1\",\"org.apache.log4j.ConsoleAp
pender\");\n");
//          out.append("        \n");
//          out.append("\t// A1 uses PatternLayout.\n");
//
      out.append("\tprops.put(\"log4j.appender.A1.layout\",\"org.apache.log4j.Pa
tternLayout\");\n");
//
      out.append("\tprops.put(\"log4j.appender.A1.layout.ConversionPattern\",\"=
*=>%-4r [%t] %-5p %c %x - %m%n\");\n");
//          out.append("\t//PropertyConfigurator.configure(props);\n");
            out.append("\n");
            out.append("\n");
            out.append("\n");
            out.append("    SourceFile sourceFile = new SourceFile();    \n");
            out.append("    TokenI token;\n");
            out.append("    \n");
            if (sourceFilename==null)
                  out.append("    "+scannerFilename+" s = new
"+scannerFilename+"(sourceFile.openFile());\n");
            else
                  out.append("    "+scannerFilename+" s = new
"+scannerFilename+"(sourceFile.openFile(\""+encodeJavaString(sourceFilename)+"\"
));\n");
            out.append("\n");

      out.append("//http://java.sun.com/j2se/1.4.2/docs/api/java/util/regex/Patt
ern.html#sum\n");
            out.append("    \n");
            out.append("    do{\n");
            out.append("      token = s.scan();\n");
            out.append("      if (token.getKind()>=-1)\n");
            out.append("      \tSystem.out.println(\"Line: \" + token.getLine()
+ \", spelling = [\" + token.getSpelling() + \"], \" + \"kind = \" +
token.getKind());\n");
            out.append("    }while(token.getKind() != TokenI.EOT);\n");
            out.append("  }\n");
            out.append("}\n");

            return new
GeneratedFile(scannerFilename+".java",packageName,out.toString());
```

```
	}


	public static void main(String args[]) throws Exception{
		String lexiconFilename = null;
		String sourceFilename = null;
		String path = "C:\\Documents and
Settings\\jfoure\\Desktop\\cs605\\Thesis\\ebnf\\";
		String packageName = "ebnftokengenerator";
		String tokenFilename = "TestToken2";
		String scannerFilename = "TestScanner2";

		System.out.println("Usage: java TokenGenerator lexiconFilename
sourceFilename path packageName tokenFilename scannerFilename");
		System.out.println("Sample: java TokenGenerator "+lexiconFilename+"
"+sourceFilename+" "+path+" "+packageName+" "+tokenFilename+"
"+scannerFilename);
		if (args.length>5){
			lexiconFilename = args[0];
			sourceFilename = args[1];
			path = args[2];
			packageName = args[3];
			tokenFilename = args[4];
			scannerFilename = args[5];
		}



		Properties props = new Properties();
		//Set root logger level to DEBUG and its only appender to A1.
		props.put("log4j.rootLogger","DEBUG,A1");
		//A1 is set to be a ConsoleAppender.
		props.put("log4j.appender.A1","org.apache.log4j.ConsoleAppender");
		// A1 uses PatternLayout.

	props.put("log4j.appender.A1.layout","org.apache.log4j.PatternLayout");
		props.put("log4j.appender.A1.layout.ConversionPattern","=*=>%-4r
[%t] %-5p %c %x - %m%n");
		//PropertyConfigurator.configure(props);

		String settings =
			"#TOKENS\n"+
			"\n"+
			//"IDENTIFIER:Identifier\n"+
			//"INTEGERLITERAL:IntegerLiteral\n"+
			//"CHARACTERLITERAL:CharacterLiteral\n"+
			"NOTUSED:\"\\a\"\n"+
			//"OPERATOR:Operator\n"+
			"ARRAY:\"array\"\n"+
			"BEGIN:\"begin\"\n";

		TokenGenerator.load(settings);
		System.out.println(GenericToken.constantSpellingMap);
		System.out.println(GenericToken.variableSpellingMap);
```

```
            TokenSourceFile tokenSourceFile = new TokenSourceFile();
            BufferedReader in = null;
            if (lexiconFilename!=null)
                  in = tokenSourceFile.openFile(lexiconFilename);
            else
                  in = tokenSourceFile.openFile();
            List tokenList = parseTokenRuleList(in);



            //-------------------- Token -----------------\\

      GeneratedFile tokenFile = generateToken(tokenFilename, packageName,
tokenList);
      tokenFile.writeOutput();

            // ----------------------- SCANNER ----------------------\\
            GeneratedFile scannerFile = generateScanner(tokenFilename,
packageName, sourceFilename, tokenList);
            scannerFile.writeOutput();




            SourceFile sourceFile = new SourceFile();
            TokenI token;
            List tokenRuleList = new ArrayList();
            int tokenCount = 0;
            int ignoredTokenCount = -2;
            Iterator tokenListIterator = tokenList.iterator();
            for (int i=1;tokenListIterator.hasNext();i++){
                  TokenEntry tokenEntry = (TokenEntry)tokenListIterator.next();
                  if (tokenEntry.ignored)
                        tokenRuleList.add(new
TokenRule(tokenEntry.variableName,((TokenLiteralDefinition)tokenEntry.tokenDefin
ition).rule,tokenEntry.isReservedWord(),ignoredTokenCount--
,tokenEntry.ignored));
                  else
                        tokenRuleList.add(new
TokenRule(tokenEntry.variableName,((TokenLiteralDefinition)tokenEntry.tokenDefin
ition).rule,tokenEntry.isReservedWord(),tokenCount++,tokenEntry.ignored));
            }

            JoshScanner s = null;
            if (sourceFilename!=null)
                  s = new
JoshScanner(sourceFile.openFile(sourceFilename),tokenRuleList);
            else
                  s = new JoshScanner(sourceFile.openFile(),tokenRuleList);
            do{
              token = s.scan();
              if (token.getKind()>=-1)
                  System.out.println("Line: " + token.getLine() + ", spelling =
[" + token.getSpelling() + "], " + "kind = " + token.getKind());
            }while(token.getKind() != TokenI.EOT);

      }
```

```
}
```

ebnftokengenerator.TokenLiteralDefinition.java

```
/*
 * Created on Aug 2, 2004
 */
package ebnftokengenerator;

/**
 * @author jfoure
 */
public class TokenLiteralDefinition extends TokenDefinition {
	String rule = null;
	public TokenLiteralDefinition(String rule){
		this.rule = rule;
	}

	public String toString(){
		return "TokenLiteralDefinition["+rule+"]";
	}

	/**
	 * @return
	 */
	public String getRule() {
		return rule;
	}

	/**
	 * @param string
	 */
	public void setRule(String string) {
		rule = string;
	}

}
```

ebnftokengenerator.TokenSourceFile.java

```
package ebnftokengenerator;
import java.io.*;
public class TokenSourceFile{

	public BufferedReader openFile(String fileName){
		BufferedReader inFile=null;
		BufferedReader stdin = new BufferedReader(new
InputStreamReader(System.in));
//		  System.out.print("Source file = ");
		System.out.flush();
		try{
//			fileName = stdin.readLine();
```

```
                inFile = new BufferedReader(new FileReader(fileName));


            }
            catch(FileNotFoundException e){
              System.out.println("The source file " + fileName + " was not
found.");
            }
            return inFile;
      }

  public BufferedReader openFile(){
    String fileName="C:\\Documents and
Settings\\jfoure\\Desktop\\cs605\\Thesis\\ebnf\\common\\emtlexicon.txt";
    return openFile(fileName);

 }
}
```

ebnftokengenerator.TokenVariableDefinition.java

```
/*
 * Created on Aug 2, 2004
 */
package ebnftokengenerator;

/**
 * @author jfoure
 */
public class TokenVariableDefinition extends TokenDefinition {
      String variableName = null;
      public TokenVariableDefinition(String variableName){
            this.variableName = variableName;
      }

      public String toString(){
            return "TokenVariableDefinition["+variableName+"]";
      }

}
```

ebnftokengenerator.TokenVariableEntry.java

```
/*
 * Created on Aug 2, 2004
 */
package ebnftokengenerator;

/**
 * @author jfoure
 */
public class TokenVariableEntry {
```

```
	String variableName = null;
	String rule = null;

	public TokenVariableEntry(String variableName, String rule){
		this.variableName = variableName;
		this.rule = rule;
	}

	public String toString(){
		return "TokenVariableEntry["+variableName+","+rule+"]";
	}

}
```

Appendix J – Ouput of Parser Generator for ML EBNF

compile.bat

```
javac -classpath .;log4j-1.2rc1.jar;common.jar mlparser/Main.java
```

run.bat

```
java -classpath .;log4j-1.2rc1.jar;common.jar mlparser.Main
```

common.Error.java

```
package common;
public class Error{
  public Error(String message, int line){
    System.out.println("Line " + line + ": " + message);
    System.exit(0);
  }
}
```

common.GenericToken.java

```
/*
 * Created on Apr 3, 2004
 */
package common;
import java.io.*;
import java.util.*;

import ebnftokengenerator.*;

/**
 * @author jfoure
 */
public class GenericToken implements TokenI{
      public static final Map constantSpellingMap = new HashMap(); public static
final Map variableSpellingMap = new HashMap();
      public int kind;
      public String spelling;
      public int line;

      public static List tokenList = null;
      public GenericToken(){

      }

  public GenericToken(int kind, String spelling, int line){
        this.kind = kind;
        this.spelling = spelling;
```

```
        this.line = line;
      }
      public String toString(){
            return "Token line: "+line+" kind: "+kind+" spelling: "+spelling;
      }



      public String lookupConstantSpelling(String spelling){
            Iterator tokenListIterator = tokenList.iterator();
            while (tokenListIterator.hasNext()){
                  TokenEntry entry = (TokenEntry) tokenListIterator.next();
                  if (entry.isReservedWord() &&
spelling.equals(((TokenLiteralDefinition)entry.getTokenDefinition()).getRule()))
                        return entry.getVariableName();
                  if (!entry.isReservedWord() &&
spelling.equals(entry.getInitialDefinition()))
                        return entry.getVariableName();
            }

      System.err.println("]]]]]]lookupConstantSpelling]"+spelling+"/null");
            return null;
      }
      public String lookupVariableSpelling(String spelling){
            Iterator tokenListIterator = tokenList.iterator();
            while (tokenListIterator.hasNext()){
                  TokenEntry entry = (TokenEntry) tokenListIterator.next();
                  if (entry.isReservedWord() &&
spelling.equals(((TokenLiteralDefinition)entry.getTokenDefinition()).getRule()))
                        return "_"+entry.getVariableName().toLowerCase();
                  if (!entry.isReservedWord() &&
spelling.equals(entry.getInitialDefinition()))
                        return "_"+entry.getVariableName().toLowerCase();
            }

      System.err.println("]]]]]]lookupVariableSpelling]"+spelling+"/null");
            return null;
      }

/*
      public String lookupConstantSpelling(String spelling){
            if (constantSpellingMap.get(spelling)==null)

      System.err.println("]]]]]]lookupConstantSpelling]"+spelling+"/"+constantSp
ellingMap.get(spelling));
            return (String)constantSpellingMap.get(spelling);
      }
      public String lookupVariableSpelling(String spelling){
            if (variableSpellingMap.get(spelling)==null)

      System.err.println("]]]]]]lookupVariableSpelling]"+spelling+"/"+variableSp
ellingMap.get(spelling));
            return (String)variableSpellingMap.get(spelling);
      }
*/

      public static void main(String args[]) throws Exception{
```

```
            /*
            String settings =
                  ", COMMA\r\n"+
                  "; SEMICOLON\r\n"+
                  "CharacterLiteral CHARLITERAL\n"+
                  "array ARRAY";
            */

      }

      /**
       * @return
       */
      public int getKind() {
            return kind;
      }

      /**
       * @return
       */
      public int getLine() {
            return line;
      }

      /**
       * @return
       */
      public String getSpelling() {
            return spelling;
      }

      /**
       * @return
       */
      public static List getTokenList() {
            return tokenList;
      }

      /**
       * @param list
       */
      public static void setTokenList(List list) {
            tokenList = list;
      }

}
```

common.TokenI.java

```
/*
 * Created on Feb 29, 2004
 */
package common;

/**
```

```
 * @author jfoure
 */
public interface TokenI {
      public final static byte EOT = -1;
      public String lookupConstantSpelling(String spelling); public String
lookupVariableSpelling(String spelling);
      public int getKind();
      public String getSpelling();
      public int getLine();


}
```

common.TokenRule.java

```
/*
 * Created on May 11, 2004
 */
package common;

/**
 * @author jfoure
 */
public class TokenRule {
      private String tokenName;
      private String regularExpression;
      private boolean reservedWord;
      private int kind;
      private boolean ignored;

      public TokenRule(String tokenName, String regularExpression, boolean
reservedWord, int kind, boolean ignored){
            this.tokenName = tokenName;
            this.regularExpression = regularExpression;
            this.reservedWord = reservedWord;
            this.kind = kind;
            this.ignored = ignored;
      }

      public GenericToken createToken(String spelling, int line){
            return new GenericToken(kind,spelling,line);
      }

      /**
       * @return
       */
      public String getRegularExpression() {
            return regularExpression;
      }

      /**
       * @return
       */
      public String getTokenName() {
            return tokenName;
```

```
}

/**
 * @param string

 */
public void setRegularExpression(String string) {
      regularExpression = string;
}

/**
 * @param string
 */
public void setTokenName(String string) {
      tokenName = string;
}

/**
 * @return
 */
public int getKind() {
      return kind;
}

/**
 * @param i
 */
public void setKind(int i) {
      kind = i;
}


/**
 * @return
 */
public boolean isIgnored() {
      return ignored;
}

/**
 * @param b
 */
public void setIgnored(boolean b) {
      ignored = b;
}

/**
 * @return
 */
public boolean isReservedWord() {
      return reservedWord;
}

/**
 * @param b
 */
public void setReservedWord(boolean b) {
```

```
            reservedWord = b;
      }

}
```

ebnftokengenerator.TokenDefinition.java

```
/*
 * Created on Aug 2, 2004
 */
package ebnftokengenerator;

/**
 * @author jfoure
 */
public abstract class TokenDefinition {
      public TokenDefinition(){
      }

      public static TokenDefinition createDefinition(String definition){
            if (definition.startsWith("\"") && definition.endsWith("\"")){
                  return new
TokenLiteralDefinition(definition.substring(1,definition.length()-1));
            } else
                  return new TokenVariableDefinition(definition);

      }

}
```

ebnftokengenerator.TokenEntry.java

```
/*
 * Created on Aug 2, 2004
 */
package ebnftokengenerator;

/**
 * @author jfoure
 */
public class TokenEntry {
      String variableName = null;
      String initialDefinition = null;
      TokenDefinition tokenDefinition = null;
      boolean ignored = false;
      boolean reservedWord = true;

      public TokenEntry(String variableName, String initialDefinition,
TokenDefinition tokenDefinition, boolean ignored){
            this.variableName = variableName;
            this.initialDefinition = initialDefinition;
            this.tokenDefinition = tokenDefinition;
```

```
            this.ignored = ignored;
      }

      public String toString(){
            return
"TokenEntry["+variableName+","+tokenDefinition+","+ignored+"]";
      }

      /**
       * @return
       */
      public boolean isIgnored() {
            return ignored;
      }

      /**
       * @return
       */
      public TokenDefinition getTokenDefinition() {
            return tokenDefinition;
      }

      /**
       * @return
       */
      public String getVariableName() {
            return variableName;
      }

      /**
       * @param b
       */
      public void setIgnored(boolean b) {
            ignored = b;
      }

      /**
       * @param definition
       */
      public void setTokenDefinition(TokenDefinition definition) {
            tokenDefinition = definition;
      }

      /**
       * @param string
       */
      public void setVariableName(String string) {
            variableName = string;
      }

      /**
       * @return
       */
      public boolean isReservedWord() {
            return reservedWord;
      }
```

```
	/**
	 * @param b
	 */
	public void setReservedWord(boolean b) {
		reservedWord = b;
	}

	/**
	 * @return
	 */
	public String getInitialDefinition() {
		return initialDefinition;
	}

	/**
	 * @param string
	 */
	public void setInitialDefinition(String string) {
		initialDefinition = string;
	}

}
```

ebnftokengenerator.TokenLiteralDefinition.java

```
/*
 * Created on Aug 2, 2004
 */
package ebnftokengenerator;

/**
 * @author jfoure
 */
public class TokenLiteralDefinition extends TokenDefinition {
	String rule = null;
	public TokenLiteralDefinition(String rule){
		this.rule = rule;
	}

	public String toString(){
		return "TokenLiteralDefinition["+rule+"]";
	}

	/**
	 * @return
	 */
	public String getRule() {
		return rule;
	}

	/**
	 * @param string
	 */
	public void setRule(String string) {
		rule = string;
```

```
	}

}
```

ebnftokengenerator.TokenVariableDefinition.java

```
/*
 * Created on Aug 2, 2004
 */
package ebnftokengenerator;

/**
 * @author jfoure
 */
public class TokenVariableDefinition extends TokenDefinition {
	String variableName = null;
	public TokenVariableDefinition(String variableName){
		this.variableName = variableName;
	}

	public String toString(){
		return "TokenVariableDefinition["+variableName+"]";
	}

}
```

mlast.AddingOperator.java

```
package mlast;
public class AddingOperator extends Ast{
public String spelling;
public int line;
public AddingOperator(String Spelling, int line){
  this.spelling = Spelling;
  this.line = line;
}
public Object visit(Visitor v, Object arg){
  return v.visitAddingOperator(this, arg, line);
}
}
```

mlast.ArgList.java

```
package mlast;
public class ArgList{
	public Expression expression;
	public ArgList_commaZeroOrMore argList_commaZeroOrMore;
	public int line;
	public ArgList(Expression expression, ArgList_commaZeroOrMore
argList_commaZeroOrMore, int line){
```

```
            this.expression = expression;
            this.argList_commaZeroOrMore = argList_commaZeroOrMore;
            this.line = line;
      }

      public String toString(){
            return "ArgList[ "+expression+","+argList_commaZeroOrMore+"]";
      }

            public Object visit(Visitor v, Object arg){
      return v.visitArgList(this, arg, line);
      }

}
```

mlast.ArgList_commaZeroOrMore.java

```
package mlast;
public class ArgList_commaZeroOrMore{
      public String _commaSpelling;
      public Expression expression;
      public ArgList_commaZeroOrMore argList_commaZeroOrMore;
      public int line;
      public ArgList_commaZeroOrMore(String _commaSpelling, Expression
expression, ArgList_commaZeroOrMore argList_commaZeroOrMore, int line){
            this._commaSpelling = _commaSpelling;
            this.expression = expression;
            this.argList_commaZeroOrMore = argList_commaZeroOrMore;
            this.line = line;
      }

      public String toString(){
            return "ArgList_commaZeroOrMore[
"+_commaSpelling+","+expression+","+argList_commaZeroOrMore+"]";
      }

            public Object visit(Visitor v, Object arg){
      return v.visitArgList_commaZeroOrMore(this, arg, line);
      }

}
```

mlast.Ast.java

```
package mlast;
public abstract class Ast{
      public abstract Object visit(Visitor v, Object arg);
}
```

mlast.BasicExpression.java

```
package mlast;
public class BasicExpression{
      public PrimaryExpression primaryExpression;
      public BasicExpressionPrimaryOperatorZeroOrMore
basicExpressionPrimaryOperatorZeroOrMore;
      public int line;
      public BasicExpression(PrimaryExpression primaryExpression,
BasicExpressionPrimaryOperatorZeroOrMore
basicExpressionPrimaryOperatorZeroOrMore, int line){
            this.primaryExpression = primaryExpression;
            this.basicExpressionPrimaryOperatorZeroOrMore =
basicExpressionPrimaryOperatorZeroOrMore;
            this.line = line;
      }

      public String toString(){
            return "BasicExpression[
"+primaryExpression+","+basicExpressionPrimaryOperatorZeroOrMore+"]";
      }

            public Object visit(Visitor v, Object arg){
      return v.visitBasicExpression(this, arg, line);
      }

}
```

mlast.BasicExpressionPrimaryOperatorZeroOrMore.java

```
package mlast;
public class BasicExpressionPrimaryOperatorZeroOrMore{
      public PrimaryOperator primaryOperator;
      public PrimaryExpression primaryExpression;
      public BasicExpressionPrimaryOperatorZeroOrMore
basicExpressionPrimaryOperatorZeroOrMore;
      public int line;
      public BasicExpressionPrimaryOperatorZeroOrMore(PrimaryOperator
primaryOperator, PrimaryExpression primaryExpression,
BasicExpressionPrimaryOperatorZeroOrMore
basicExpressionPrimaryOperatorZeroOrMore, int line){
            this.primaryOperator = primaryOperator;
            this.primaryExpression = primaryExpression;
            this.basicExpressionPrimaryOperatorZeroOrMore =
basicExpressionPrimaryOperatorZeroOrMore;
            this.line = line;
      }

      public String toString(){
            return "BasicExpressionPrimaryOperatorZeroOrMore[
"+primaryOperator+","+primaryExpression+","+basicExpressionPrimaryOperatorZeroOr
More+"]";
      }

            public Object visit(Visitor v, Object arg){
      return v.visitBasicExpressionPrimaryOperatorZeroOrMore(this, arg, line);
```

```
	}

}
```

mlast.Call.java

```
package mlast;
public class Call{
	public Identifier identifier;
	public String _lparenSpelling;
	public CallArgListZeroOrOne callArgListZeroOrOne;
	public String _rparenSpelling;
	public int line;

	public Call(Identifier identifier, String _lparenSpelling,
CallArgListZeroOrOne callArgListZeroOrOne, String _rparenSpelling, int line){
		this.identifier = identifier;
		this._lparenSpelling = _lparenSpelling;
		this.callArgListZeroOrOne = callArgListZeroOrOne;
		this._rparenSpelling = _rparenSpelling;
		this.line = line;
	}

	public String toString(){
		return "Call[
"+identifier+","+_lparenSpelling+","+callArgListZeroOrOne+","+_rparenSpelling+"]
";
	}

		public Object visit(Visitor v, Object arg){
	return v.visitCall(this, arg, line);
	}

}
```

mlast.CallArgListZeroOrOne.java

```
package mlast;
public class CallArgListZeroOrOne{
	public ArgList argList;
	public int line;

	public CallArgListZeroOrOne(ArgList argList, int line){
		this.argList = argList;
		this.line = line;
	}

	public String toString(){
		return "CallArgListZeroOrOne[ "+argList+"]";
	}

		public Object visit(Visitor v, Object arg){
```

```
      return v.visitCallArgListZeroOrOne(this, arg, line);
      }

}
```

mlast.ConditionalExpression.java

```
package mlast;
public class ConditionalExpression{
      public String _ifSpelling;
      public BasicExpression basicExpression;
      public String _thenSpelling;
      public Expression expression;
      public String _elseSpelling;
      public Expression expression2;
      public int line;

      public ConditionalExpression(String _ifSpelling, BasicExpression
basicExpression, String _thenSpelling, Expression expression, String
_elseSpelling, Expression expression2, int line){
            this._ifSpelling = _ifSpelling;
            this.basicExpression = basicExpression;
            this._thenSpelling = _thenSpelling;
            this.expression = expression;
            this._elseSpelling = _elseSpelling;
            this.expression2 = expression2;
            this.line = line;
      }

      public String toString(){
            return "ConditionalExpression[
"+_ifSpelling+","+basicExpression+","+_thenSpelling+","+expression+","+_elseSpel
ling+","+expression2+"]";
      }

            public Object visit(Visitor v, Object arg){
      return v.visitConditionalExpression(this, arg, line);
      }

}
```

mlast.Display.java

```
package mlast;
public class Display{
      final private int BLANKS = 2;
      public Display(Object p){
            display(p, 0);
      }
      private static void spaces(int count){
              for(int i = 1; i <= count; i++)
                        System.out.print(" ");
```

```
	}
	private void display(Object p, int count){
		if(p == null) return;
		String s = p.getClass().getName();
		spaces(count);
		if (s.equals("java.lang.String")){
			System.out.println("<"+p+">");
		} else {
			System.out.println(s);
		}
		if(s.equals("mlast.Program")){
			if (((mlast.Program)p).programFunctionsZeroOrMore!=null)
				display(((mlast.Program)p).programFunctionsZeroOrMore,
count+BLANKS);
				display(((mlast.Program)p).call, count+BLANKS);
				display(((mlast.Program)p)._semicolonSpelling,
count+BLANKS);
		}
		if(s.equals("mlast.ProgramFunctionsZeroOrMore")){
				display(((mlast.ProgramFunctionsZeroOrMore)p).functions,
count+BLANKS);

	display(((mlast.ProgramFunctionsZeroOrMore)p)._semicolonSpelling,
count+BLANKS);
			if
(((mlast.ProgramFunctionsZeroOrMore)p).programFunctionsZeroOrMore!=null)

	display(((mlast.ProgramFunctionsZeroOrMore)p).programFunctionsZeroOrMore,
count+BLANKS);
		}
		if(s.equals("mlast.Functions")){
				display(((mlast.Functions)p)._funSpelling,
count+BLANKS);
				display(((mlast.Functions)p).identifier, count+BLANKS);
				display(((mlast.Functions)p)._lparenSpelling,
count+BLANKS);
			if (((mlast.Functions)p).functionsParListZeroOrOne!=null)
				display(((mlast.Functions)p).functionsParListZeroOrOne,
count+BLANKS);
				display(((mlast.Functions)p)._rparenSpelling,
count+BLANKS);
				display(((mlast.Functions)p)._colonSpelling,
count+BLANKS);
				display(((mlast.Functions)p).type, count+BLANKS);
				display(((mlast.Functions)p)._equalSpelling,
count+BLANKS);
				display(((mlast.Functions)p).expression, count+BLANKS);
		}
		if(s.equals("mlast.FunctionsParListZeroOrOne")){
				display(((mlast.FunctionsParListZeroOrOne)p).parList,
count+BLANKS);
		}
		if(s.equals("mlast.ParList")){
				display(((mlast.ParList)p).par, count+BLANKS);
			if (((mlast.ParList)p).parList_commaZeroOrMore!=null)
				display(((mlast.ParList)p).parList_commaZeroOrMore,
count+BLANKS);
```

```
            }
            if(s.equals("mlast.ParList_commaZeroOrMore")){

      display(((mlast.ParList_commaZeroOrMore)p)._commaSpelling, count+BLANKS);
                        display(((mlast.ParList_commaZeroOrMore)p).par,
count+BLANKS);
                  if
(((mlast.ParList_commaZeroOrMore)p).parList_commaZeroOrMore!=null)

      display(((mlast.ParList_commaZeroOrMore)p).parList_commaZeroOrMore,
count+BLANKS);
            }
            if(s.equals("mlast.Par")){
                        display(((mlast.Par)p).identifier, count+BLANKS);
                        display(((mlast.Par)p)._colonSpelling, count+BLANKS);
                        display(((mlast.Par)p).type, count+BLANKS);
            }
            if(s.equals("mlast.Type")){
                        display(((mlast.Type)p).simpleType, count+BLANKS);
                  if (((mlast.Type)p).type_listZeroOrOne!=null)
                        display(((mlast.Type)p).type_listZeroOrOne,
count+BLANKS);
            }
            if(s.equals("mlast.Type_listZeroOrOne")){
                        display(((mlast.Type_listZeroOrOne)p)._listSpelling,
count+BLANKS);
            }
            if(s.equals("mlast.ExpressionConditionalExpression")){

      display(((mlast.ExpressionConditionalExpression)p).conditionalExpression,
count+BLANKS);
            }
            if(s.equals("mlast.ExpressionBasicExpression")){

      display(((mlast.ExpressionBasicExpression)p).basicExpression,
count+BLANKS);
            }
            if(s.equals("mlast.ConditionalExpression")){
                        display(((mlast.ConditionalExpression)p)._ifSpelling,
count+BLANKS);

      display(((mlast.ConditionalExpression)p).basicExpression, count+BLANKS);
                        display(((mlast.ConditionalExpression)p)._thenSpelling,
count+BLANKS);
                        display(((mlast.ConditionalExpression)p).expression,
count+BLANKS);
                        display(((mlast.ConditionalExpression)p)._elseSpelling,
count+BLANKS);
                        display(((mlast.ConditionalExpression)p).expression2,
count+BLANKS);
            }
            if(s.equals("mlast.BasicExpression")){
                        display(((mlast.BasicExpression)p).primaryExpression,
count+BLANKS);
                  if
(((mlast.BasicExpression)p).basicExpressionPrimaryOperatorZeroOrMore!=null)
```

```
      display(((mlast.BasicExpression)p).basicExpressionPrimaryOperatorZeroOrMor
e, count+BLANKS);
            }
            if(s.equals("mlast.BasicExpressionPrimaryOperatorZeroOrMore")){

      display(((mlast.BasicExpressionPrimaryOperatorZeroOrMore)p).primaryOperato
r, count+BLANKS);

      display(((mlast.BasicExpressionPrimaryOperatorZeroOrMore)p).primaryExpress
ion, count+BLANKS);
                  if
(((mlast.BasicExpressionPrimaryOperatorZeroOrMore)p).basicExpressionPrimaryOpera
torZeroOrMore!=null)

      display(((mlast.BasicExpressionPrimaryOperatorZeroOrMore)p).basicExpressio
nPrimaryOperatorZeroOrMore, count+BLANKS);
            }
            if(s.equals("mlast.PrimaryExpression")){
                        display(((mlast.PrimaryExpression)p).simpleExpression,
count+BLANKS);
                  if
(((mlast.PrimaryExpression)p).primaryExpressionRelationalOperatorZeroOrOne!=null
)

      display(((mlast.PrimaryExpression)p).primaryExpressionRelationalOperatorZe
roOrOne, count+BLANKS);
            }
            if(s.equals("mlast.PrimaryExpressionRelationalOperatorZeroOrOne")){

      display(((mlast.PrimaryExpressionRelationalOperatorZeroOrOne)p).relational
Operator, count+BLANKS);

      display(((mlast.PrimaryExpressionRelationalOperatorZeroOrOne)p).simpleExpr
ession, count+BLANKS);
            }
            if(s.equals("mlast.SimpleExpression")){
                  if
(((mlast.SimpleExpression)p).simpleExpression_negZeroOrOne!=null)

      display(((mlast.SimpleExpression)p).simpleExpression_negZeroOrOne,
count+BLANKS);
                        display(((mlast.SimpleExpression)p).term, count+BLANKS);
                  if
(((mlast.SimpleExpression)p).simpleExpressionAddingOperatorZeroOrMore!=null)

      display(((mlast.SimpleExpression)p).simpleExpressionAddingOperatorZeroOrMo
re, count+BLANKS);
            }
            if(s.equals("mlast.SimpleExpression_negZeroOrOne")){

      display(((mlast.SimpleExpression_negZeroOrOne)p)._negSpelling,
count+BLANKS);
            }
            if(s.equals("mlast.SimpleExpressionAddingOperatorZeroOrMore")){
```

```
      display(((mlast.SimpleExpressionAddingOperatorZeroOrMore)p).addingOperator
, count+BLANKS);

      display(((mlast.SimpleExpressionAddingOperatorZeroOrMore)p).term,
count+BLANKS);
                  if
(((mlast.SimpleExpressionAddingOperatorZeroOrMore)p).simpleExpressionAddingOpera
torZeroOrMore!=null)

      display(((mlast.SimpleExpressionAddingOperatorZeroOrMore)p).simpleExpressi
onAddingOperatorZeroOrMore, count+BLANKS);
            }
            if(s.equals("mlast.Term")){
                        display(((mlast.Term)p).factor, count+BLANKS);
                  if (((mlast.Term)p).termMultiplyingOperatorZeroOrMore!=null)

      display(((mlast.Term)p).termMultiplyingOperatorZeroOrMore, count+BLANKS);
            }
            if(s.equals("mlast.TermMultiplyingOperatorZeroOrMore")){

      display(((mlast.TermMultiplyingOperatorZeroOrMore)p).multiplyingOperator,
count+BLANKS);

      display(((mlast.TermMultiplyingOperatorZeroOrMore)p).factor,
count+BLANKS);
                  if
(((mlast.TermMultiplyingOperatorZeroOrMore)p).termMultiplyingOperatorZeroOrMore!
=null)

      display(((mlast.TermMultiplyingOperatorZeroOrMore)p).termMultiplyingOperat
orZeroOrMore, count+BLANKS);
            }
            if(s.equals("mlast.Factor_integerliteral")){
                        display(((mlast.Factor_integerliteral)p).integerLiteral,
count+BLANKS);
            }
            if(s.equals("mlast.Factor_stringliteral")){
                        display(((mlast.Factor_stringliteral)p).stringLiteral,
count+BLANKS);
            }
            if(s.equals("mlast.Factor_identifier")){
                        display(((mlast.Factor_identifier)p).identifier,
count+BLANKS);
                  if
(((mlast.Factor_identifier)p).factor_identifier_lparenZeroOrOne!=null)

      display(((mlast.Factor_identifier)p).factor_identifier_lparenZeroOrOne,
count+BLANKS);
            }
            if(s.equals("mlast.Factor_identifier_lparenZeroOrOne")){

      display(((mlast.Factor_identifier_lparenZeroOrOne)p)._lparenSpelling,
count+BLANKS);
                  if
(((mlast.Factor_identifier_lparenZeroOrOne)p).factor_identifier_lparenZeroOrOneA
rgListZeroOrOne!=null)
```

```
      display(((mlast.Factor_identifier_lparenZeroOrOne)p).factor_identifier_lpa
renZeroOrOneArgListZeroOrOne, count+BLANKS);

      display(((mlast.Factor_identifier_lparenZeroOrOne)p)._rparenSpelling,
count+BLANKS);
            }

      if(s.equals("mlast.Factor_identifier_lparenZeroOrOneArgListZeroOrOne")){

      display(((mlast.Factor_identifier_lparenZeroOrOneArgListZeroOrOne)p).argLi
st, count+BLANKS);
            }
            if(s.equals("mlast.FactorListLiteral")){
                        display(((mlast.FactorListLiteral)p).listLiteral,
count+BLANKS);
            }
            if(s.equals("mlast.Factor_lparen")){
                        display(((mlast.Factor_lparen)p)._lparenSpelling,
count+BLANKS);
                        display(((mlast.Factor_lparen)p).expression,
count+BLANKS);
                        display(((mlast.Factor_lparen)p)._rparenSpelling,
count+BLANKS);
            }
            if(s.equals("mlast.Factor_not")){
                        display(((mlast.Factor_not)p)._notSpelling,
count+BLANKS);
                        display(((mlast.Factor_not)p).factor, count+BLANKS);
            }
            if(s.equals("mlast.ListLiteral")){
                        display(((mlast.ListLiteral)p)._lbracketSpelling,
count+BLANKS);
                  if
(((mlast.ListLiteral)p).listLiteralExpressionListZeroOrOne!=null)

      display(((mlast.ListLiteral)p).listLiteralExpressionListZeroOrOne,
count+BLANKS);
                        display(((mlast.ListLiteral)p)._rbracketSpelling,
count+BLANKS);
            }
            if(s.equals("mlast.ListLiteralExpressionListZeroOrOne")){

      display(((mlast.ListLiteralExpressionListZeroOrOne)p).expressionList,
count+BLANKS);
            }
            if(s.equals("mlast.ExpressionList")){
                        display(((mlast.ExpressionList)p).expression,
count+BLANKS);
                  if
(((mlast.ExpressionList)p).expressionList_commaZeroOrMore!=null)

      display(((mlast.ExpressionList)p).expressionList_commaZeroOrMore,
count+BLANKS);
            }
            if(s.equals("mlast.ExpressionList_commaZeroOrMore")){
```

```
      display(((mlast.ExpressionList_commaZeroOrMore)p)._commaSpelling,
count+BLANKS);

      display(((mlast.ExpressionList_commaZeroOrMore)p).expression,
count+BLANKS);
                  if
(((mlast.ExpressionList_commaZeroOrMore)p).expressionList_commaZeroOrMore!=null)

      display(((mlast.ExpressionList_commaZeroOrMore)p).expressionList_commaZero
OrMore, count+BLANKS);
            }
            if(s.equals("mlast.Call")){
                        display(((mlast.Call)p).identifier, count+BLANKS);
                        display(((mlast.Call)p)._lparenSpelling, count+BLANKS);
                  if (((mlast.Call)p).callArgListZeroOrOne!=null)
                        display(((mlast.Call)p).callArgListZeroOrOne,
count+BLANKS);
                        display(((mlast.Call)p)._rparenSpelling, count+BLANKS);
            }
            if(s.equals("mlast.CallArgListZeroOrOne")){
                        display(((mlast.CallArgListZeroOrOne)p).argList,
count+BLANKS);
            }
            if(s.equals("mlast.ArgList")){
                        display(((mlast.ArgList)p).expression, count+BLANKS);
                  if (((mlast.ArgList)p).argList_commaZeroOrMore!=null)
                        display(((mlast.ArgList)p).argList_commaZeroOrMore,
count+BLANKS);
            }
            if(s.equals("mlast.ArgList_commaZeroOrMore")){

      display(((mlast.ArgList_commaZeroOrMore)p)._commaSpelling, count+BLANKS);
                        display(((mlast.ArgList_commaZeroOrMore)p).expression,
count+BLANKS);
                  if
(((mlast.ArgList_commaZeroOrMore)p).argList_commaZeroOrMore!=null)

      display(((mlast.ArgList_commaZeroOrMore)p).argList_commaZeroOrMore,
count+BLANKS);
            }
            if(s.equals("mlast.Identifier")){
                  spaces(count);
                  System.out.println("***"+((Identifier)p).spelling+"***");
            }
            if(s.equals("mlast.IntegerLiteral")){
                  spaces(count);
                  System.out.println("***"+((IntegerLiteral)p).spelling+"***");
            }
            if(s.equals("mlast.StringLiteral")){
                  spaces(count);
                  System.out.println("***"+((StringLiteral)p).spelling+"***");
            }
      }
}
```

mlast.Expression.java

```
package mlast;
public abstract class Expression extends Ast{}
```

mlast.ExpressionBasicExpression.java

```
package mlast;
public class ExpressionBasicExpression extends Expression{
      public BasicExpression basicExpression;
      public int line;

      public ExpressionBasicExpression(BasicExpression basicExpression, int
line){
            this.basicExpression = basicExpression;
            this.line = line;
      }

      public String toString(){
            return "ExpressionBasicExpression[ "+basicExpression+"]";
      }

            public Object visit(Visitor v, Object arg){
      return v.visitExpressionBasicExpression(this, arg, line);
      }

}
```

mlast.ExpressionConditionalExpression.java

```
package mlast;
public class ExpressionConditionalExpression extends Expression{
      public ConditionalExpression conditionalExpression;
      public int line;

      public ExpressionConditionalExpression(ConditionalExpression
conditionalExpression, int line){
            this.conditionalExpression = conditionalExpression;
            this.line = line;
      }

      public String toString(){
            return "ExpressionConditionalExpression[
"+conditionalExpression+"]";
      }

            public Object visit(Visitor v, Object arg){
      return v.visitExpressionConditionalExpression(this, arg, line);
      }

}
```

mlast.ExpressionList.java

```
package mlast;
public class ExpressionList{
      public Expression expression;
      public ExpressionList_commaZeroOrMore expressionList_commaZeroOrMore;
      public int line;
      public ExpressionList(Expression expression,
ExpressionList_commaZeroOrMore expressionList_commaZeroOrMore, int line){
            this.expression = expression;
            this.expressionList_commaZeroOrMore =
expressionList_commaZeroOrMore;
            this.line = line;
      }

      public String toString(){
            return "ExpressionList[
"+expression+","+expressionList_commaZeroOrMore+"]";
      }

            public Object visit(Visitor v, Object arg){
      return v.visitExpressionList(this, arg, line);
      }

}
```

mlast.ExpressionList_commaZeroOrMore.java

```
package mlast;
public class ExpressionList_commaZeroOrMore{
      public String _commaSpelling;
      public Expression expression;
      public ExpressionList_commaZeroOrMore expressionList_commaZeroOrMore;
      public int line;
      public ExpressionList_commaZeroOrMore(String _commaSpelling, Expression
expression, ExpressionList_commaZeroOrMore expressionList_commaZeroOrMore, int
line){
            this._commaSpelling = _commaSpelling;
            this.expression = expression;
            this.expressionList_commaZeroOrMore =
expressionList_commaZeroOrMore;
            this.line = line;
      }

      public String toString(){
            return "ExpressionList_commaZeroOrMore[
"+_commaSpelling+","+expression+","+expressionList_commaZeroOrMore+"]";
      }

            public Object visit(Visitor v, Object arg){
      return v.visitExpressionList_commaZeroOrMore(this, arg, line);
      }
```

```
}
```

mlast.Factor.java

```
package mlast;
public abstract class Factor extends Ast{}
```

mlast.FactorListLiteral.java

```
package mlast;
public class FactorListLiteral extends Factor{
      public ListLiteral listLiteral;
      public int line;

      public FactorListLiteral(ListLiteral listLiteral, int line){
            this.listLiteral = listLiteral;
            this.line = line;
      }

      public String toString(){
            return "FactorListLiteral[ "+listLiteral+"]";
      }

            public Object visit(Visitor v, Object arg){
      return v.visitFactorListLiteral(this, arg, line);
      }

}
```

mlast.Factor_identifier.java

```
package mlast;
public class Factor_identifier extends Factor{
      public Identifier identifier;
      public Factor_identifier_lparenZeroOrOne
factor_identifier_lparenZeroOrOne;
      public int line;

      public Factor_identifier(Identifier identifier,
Factor_identifier_lparenZeroOrOne factor_identifier_lparenZeroOrOne, int line){
            this.identifier = identifier;
            this.factor_identifier_lparenZeroOrOne =
factor_identifier_lparenZeroOrOne;
            this.line = line;
      }

      public String toString(){
            return "Factor_identifier[
"+identifier+","+factor_identifier_lparenZeroOrOne+"]";
      }
```

```
            public Object visit(Visitor v, Object arg){
      return v.visitFactor_identifier(this, arg, line);
      }

}
```

mlast.Factor_identifier_lparenZeroOrOne.java

```
package mlast;
public class Factor_identifier_lparenZeroOrOne{
      public String _lparenSpelling;
      public Factor_identifier_lparenZeroOrOneArgListZeroOrOne
factor_identifier_lparenZeroOrOneArgListZeroOrOne;
      public String _rparenSpelling;
      public int line;

      public Factor_identifier_lparenZeroOrOne(String _lparenSpelling,
Factor_identifier_lparenZeroOrOneArgListZeroOrOne
factor_identifier_lparenZeroOrOneArgListZeroOrOne, String _rparenSpelling, int
line){
            this._lparenSpelling = _lparenSpelling;
            this.factor_identifier_lparenZeroOrOneArgListZeroOrOne =
factor_identifier_lparenZeroOrOneArgListZeroOrOne;
            this._rparenSpelling = _rparenSpelling;
            this.line = line;
      }

      public String toString(){
            return "Factor_identifier_lparenZeroOrOne[
"+_lparenSpelling+","+factor_identifier_lparenZeroOrOneArgListZeroOrOne+","+_rpa
renSpelling+"]";
      }

            public Object visit(Visitor v, Object arg){
      return v.visitFactor_identifier_lparenZeroOrOne(this, arg, line);
      }

}
```

mlast.Factor_identifier_lparenZeroOrOneArgListZeroOrOne.java

```
package mlast;
public class Factor_identifier_lparenZeroOrOneArgListZeroOrOne{
      public ArgList argList;
      public int line;

      public Factor_identifier_lparenZeroOrOneArgListZeroOrOne(ArgList argList,
int line){
            this.argList = argList;
            this.line = line;
      }
```

```
	public String toString(){
		return "Factor_identifier_lparenZeroOrOneArgListZeroOrOne[
"+argList+"]";
	}

		public Object visit(Visitor v, Object arg){
	return v.visitFactor_identifier_lparenZeroOrOneArgListZeroOrOne(this, arg,
line);
	}

}
```

mlast.Factor_integerliteral.java

```
package mlast;
public class Factor_integerliteral extends Factor{
	public IntegerLiteral integerLiteral;
	public int line;

	public Factor_integerliteral(IntegerLiteral integerLiteral, int line){
		this.integerLiteral = integerLiteral;
		this.line = line;
	}

	public String toString(){
		return "Factor_integerliteral[ "+integerLiteral+"]";
	}

		public Object visit(Visitor v, Object arg){
	return v.visitFactor_integerliteral(this, arg, line);
	}

}
```

mlast.Factor_lparen.java

```
package mlast;
public class Factor_lparen extends Factor{
	public String _lparenSpelling;
	public Expression expression;
	public String _rparenSpelling;
	public int line;

	public Factor_lparen(String _lparenSpelling, Expression expression, String
_rparenSpelling, int line){
		this._lparenSpelling = _lparenSpelling;
		this.expression = expression;
		this._rparenSpelling = _rparenSpelling;
		this.line = line;
	}
```

```
	public String toString(){
		return "Factor_lparen[
"+_lparenSpelling+","+expression+","+_rparenSpelling+"]";
	}

		public Object visit(Visitor v, Object arg){
	return v.visitFactor_lparen(this, arg, line);
	}

}
```

mlast.Factor_not.java

```
package mlast;


public class Factor_not extends Factor{
	public String _notSpelling;
	public Factor factor;
	public int line;

	public Factor_not(String _notSpelling, Factor factor, int line){
		this._notSpelling = _notSpelling;
		this.factor = factor;
		this.line = line;
	}

	public String toString(){
		return "Factor_not[ "+_notSpelling+","+factor+"]";
	}

		public Object visit(Visitor v, Object arg){
	return v.visitFactor_not(this, arg, line);
	}

}
```

mlast.Factor_stringliteral.java

```
package mlast;
public class Factor_stringliteral extends Factor{
	public StringLiteral stringLiteral;
	public int line;

	public Factor_stringliteral(StringLiteral stringLiteral, int line){
		this.stringLiteral = stringLiteral;
		this.line = line;
	}

	public String toString(){
		return "Factor_stringliteral[ "+stringLiteral+"]";
	}
```

```
            public Object visit(Visitor v, Object arg){
      return v.visitFactor_stringliteral(this, arg, line);
      }

}
```

mlast.Functions.java

```
package mlast;
public class Functions{
      public String _funSpelling;
      public Identifier identifier;
      public String _lparenSpelling;
      public FunctionsParListZeroOrOne functionsParListZeroOrOne;
      public String _rparenSpelling;
      public String _colonSpelling;
      public Type type;
      public String _equalSpelling;
      public Expression expression;
      public int line;

      public Functions(String _funSpelling, Identifier identifier, String
_lparenSpelling, FunctionsParListZeroOrOne functionsParListZeroOrOne, String
_rparenSpelling, String _colonSpelling, Type type, String _equalSpelling,
Expression expression, int line){
            this._funSpelling = _funSpelling;
            this.identifier = identifier;
            this._lparenSpelling = _lparenSpelling;
            this.functionsParListZeroOrOne = functionsParListZeroOrOne;
            this._rparenSpelling = _rparenSpelling;
            this._colonSpelling = _colonSpelling;
            this.type = type;
            this._equalSpelling = _equalSpelling;
            this.expression = expression;
            this.line = line;
      }

      public String toString(){
            return "Functions[
"+_funSpelling+","+identifier+","+_lparenSpelling+","+functionsParListZeroOrOne+
","+_rparenSpelling+","+_colonSpelling+","+type+","+_equalSpelling+","+expressio
n+"]";
      }

            public Object visit(Visitor v, Object arg){
      return v.visitFunctions(this, arg, line);
      }

}
```

mlast.FunctionsParListZeroOrOne.java

```
package mlast;
public class FunctionsParListZeroOrOne{
      public ParList parList;
      public int line;

      public FunctionsParListZeroOrOne(ParList parList, int line){
            this.parList = parList;
            this.line = line;
      }

      public String toString(){
            return "FunctionsParListZeroOrOne[ "+parList+"]";
      }

            public Object visit(Visitor v, Object arg){
      return v.visitFunctionsParListZeroOrOne(this, arg, line);
      }

}
```

mlast.Identifier.java

```
package mlast;
public class Identifier{
      public String spelling;
      public int line;

      public Identifier(String spelling, int line){
            this.spelling = spelling;
            this.line = line;
      }

      public String toString(){
            return "Identifier[ "+spelling+" ]";
      }

            public Object visit(Visitor v, Object arg){
      return v.visitIdentifier(this, arg, line);
      }

}
```

mlast.IntegerLiteral.java

```
package mlast;
public class IntegerLiteral{
      public String spelling;
      public int line;

      public IntegerLiteral(String spelling, int line){
            this.spelling = spelling;
```

```
            this.line = line;
      }

      public String toString(){
            return "IntegerLiteral[ "+spelling+" ]";
      }

            public Object visit(Visitor v, Object arg){
      return v.visitIntegerLiteral(this, arg, line);
      }

}
```

mlast.ListLiteral.java

```
package mlast;
public class ListLiteral{
      public String _lbracketSpelling;
      public ListLiteralExpressionListZeroOrOne
listLiteralExpressionListZeroOrOne;
      public String _rbracketSpelling;

      public int line;
      public ListLiteral(String _lbracketSpelling,
ListLiteralExpressionListZeroOrOne listLiteralExpressionListZeroOrOne, String
_rbracketSpelling, int line){
            this._lbracketSpelling = _lbracketSpelling;
            this.listLiteralExpressionListZeroOrOne =
listLiteralExpressionListZeroOrOne;
            this._rbracketSpelling = _rbracketSpelling;
            this.line = line;
      }

      public String toString(){
            return "ListLiteral[
"+_lbracketSpelling+","+listLiteralExpressionListZeroOrOne+","+_rbracketSpelling
+"]";
      }

            public Object visit(Visitor v, Object arg){
      return v.visitListLiteral(this, arg, line);
      }

}
```

mlast.ListLiteralExpressionListZeroOrOne.java

```
package mlast;
public class ListLiteralExpressionListZeroOrOne{
      public ExpressionList expressionList;
      public int line;
```

```
      public ListLiteralExpressionListZeroOrOne(ExpressionList expressionList,
int line){
            this.expressionList = expressionList;
            this.line = line;
      }

      public String toString(){
            return "ListLiteralExpressionListZeroOrOne[ "+expressionList+"]";
      }

            public Object visit(Visitor v, Object arg){
      return v.visitListLiteralExpressionListZeroOrOne(this, arg, line);
      }

}
```

mlast.MultiplyingOperator.java

```
package mlast;
public class MultiplyingOperator extends Ast{
public String spelling;
public int line;
public MultiplyingOperator(String Spelling, int line){
  this.spelling = Spelling;
  this.line = line;
}
public Object visit(Visitor v, Object arg){
  return v.visitMultiplyingOperator(this, arg, line);
}
}
```

mlast.Par.java

```
package mlast;
public class Par{
      public Identifier identifier;
      public String _colonSpelling;
      public Type type;
      public int line;

      public Par(Identifier identifier, String _colonSpelling, Type type, int
line){
            this.identifier = identifier;
            this._colonSpelling = _colonSpelling;
            this.type = type;
            this.line = line;
      }

      public String toString(){
            return "Par[ "+identifier+","+_colonSpelling+","+type+"]";
      }
```

```
            public Object visit(Visitor v, Object arg){
      return v.visitPar(this, arg, line);
      }

}
```

mlast.ParList.java

```
package mlast;
public class ParList{
      public Par par;
      public ParList_commaZeroOrMore parList_commaZeroOrMore;
      public int line;
      public ParList(Par par, ParList_commaZeroOrMore parList_commaZeroOrMore,
int line){
            this.par = par;
            this.parList_commaZeroOrMore = parList_commaZeroOrMore;
            this.line = line;
      }

      public String toString(){
            return "ParList[ "+par+","+parList_commaZeroOrMore+"]";
      }

            public Object visit(Visitor v, Object arg){
      return v.visitParList(this, arg, line);
      }

}
```

mlast.ParList_commaZeroOrMore.java

```
package mlast;
public class ParList_commaZeroOrMore{
      public String _commaSpelling;
      public Par par;
      public ParList_commaZeroOrMore parList_commaZeroOrMore;
      public int line;
      public ParList_commaZeroOrMore(String _commaSpelling, Par par,
ParList_commaZeroOrMore parList_commaZeroOrMore, int line){
            this._commaSpelling = _commaSpelling;
            this.par = par;
            this.parList_commaZeroOrMore = parList_commaZeroOrMore;
            this.line = line;
      }

      public String toString(){
            return "ParList_commaZeroOrMore[
"+_commaSpelling+","+par+","+parList_commaZeroOrMore+"]";
      }

            public Object visit(Visitor v, Object arg){
```

```
      return v.visitParList_commaZeroOrMore(this, arg, line);
      }

}
```

mlast.PrimaryExpression.java

```
package mlast;
public class PrimaryExpression{
      public SimpleExpression simpleExpression;
      public PrimaryExpressionRelationalOperatorZeroOrOne
primaryExpressionRelationalOperatorZeroOrOne;
      public int line;
      public PrimaryExpression(SimpleExpression simpleExpression,
PrimaryExpressionRelationalOperatorZeroOrOne
primaryExpressionRelationalOperatorZeroOrOne, int line){
            this.simpleExpression = simpleExpression;
            this.primaryExpressionRelationalOperatorZeroOrOne =
primaryExpressionRelationalOperatorZeroOrOne;
            this.line = line;
      }

      public String toString(){
            return "PrimaryExpression[
"+simpleExpression+","+primaryExpressionRelationalOperatorZeroOrOne+"]";
      }

            public Object visit(Visitor v, Object arg){
      return v.visitPrimaryExpression(this, arg, line);
      }

}
```

mlast.PrimaryExpressionRelationalOperatorZeroOrOne.java

```
package mlast;
public class PrimaryExpressionRelationalOperatorZeroOrOne{
      public RelationalOperator relationalOperator;
      public SimpleExpression simpleExpression;
      public int line;
      public PrimaryExpressionRelationalOperatorZeroOrOne(RelationalOperator
relationalOperator, SimpleExpression simpleExpression, int line){
            this.relationalOperator = relationalOperator;
            this.simpleExpression = simpleExpression;
            this.line = line;
      }

      public String toString(){
            return "PrimaryExpressionRelationalOperatorZeroOrOne[
"+relationalOperator+","+simpleExpression+"]";
      }
```

```
            public Object visit(Visitor v, Object arg){
      return v.visitPrimaryExpressionRelationalOperatorZeroOrOne(this, arg,
line);
      }

}
```

mlast.PrimaryOperator.java

```
package mlast;
public class PrimaryOperator extends Ast{
public String spelling;
public int line;
public PrimaryOperator(String Spelling, int line){
  this.spelling = Spelling;
  this.line = line;
}
public Object visit(Visitor v, Object arg){
  return v.visitPrimaryOperator(this, arg, line);
}
}
```

mlast.Program.java

```
package mlast;
public class Program{
      public ProgramFunctionsZeroOrMore programFunctionsZeroOrMore;
      public Call call;
      public String _semicolonSpelling;
      public int line;

      public Program(ProgramFunctionsZeroOrMore programFunctionsZeroOrMore, Call
call, String _semicolonSpelling, int line){
            this.programFunctionsZeroOrMore = programFunctionsZeroOrMore;
            this.call = call;
            this._semicolonSpelling = _semicolonSpelling;
            this.line = line;
      }

      public String toString(){
            return "Program[
"+programFunctionsZeroOrMore+","+call+","+_semicolonSpelling+"]";
      }

            public Object visit(Visitor v, Object arg){
      return v.visitProgram(this, arg, line);
      }

}
```

mlast.ProgramFunctionsZeroOrMore.java

```
package mlast;
public class ProgramFunctionsZeroOrMore{
      public Functions functions;
      public String _semicolonSpelling;
      public ProgramFunctionsZeroOrMore programFunctionsZeroOrMore;
      public int line;
      public ProgramFunctionsZeroOrMore(Functions functions, String
_semicolonSpelling, ProgramFunctionsZeroOrMore programFunctionsZeroOrMore, int
line){
            this.functions = functions;
            this._semicolonSpelling = _semicolonSpelling;
            this.programFunctionsZeroOrMore = programFunctionsZeroOrMore;
            this.line = line;
      }


      public String toString(){
            return "ProgramFunctionsZeroOrMore[
"+functions+","+_semicolonSpelling+","+programFunctionsZeroOrMore+"]";
      }

            public Object visit(Visitor v, Object arg){
      return v.visitProgramFunctionsZeroOrMore(this, arg, line);
      }


}
```

mlast.RelationalOperator.java

```
package mlast;
public class RelationalOperator extends Ast{
public String spelling;
public int line;
public RelationalOperator(String Spelling, int line){
  this.spelling = Spelling;
  this.line = line;
}
public Object visit(Visitor v, Object arg){
  return v.visitRelationalOperator(this, arg, line);
}
}
```

mlast.SimpleExpression.java

```
package mlast;
public class SimpleExpression{
      public SimpleExpression_negZeroOrOne simpleExpression_negZeroOrOne;
      public Term term;
```

```
      public SimpleExpressionAddingOperatorZeroOrMore
simpleExpressionAddingOperatorZeroOrMore;
      public int line;
      public SimpleExpression(SimpleExpression_negZeroOrOne
simpleExpression_negZeroOrOne, Term term,
SimpleExpressionAddingOperatorZeroOrMore
simpleExpressionAddingOperatorZeroOrMore, int line){
            this.simpleExpression_negZeroOrOne = simpleExpression_negZeroOrOne;
            this.term = term;
            this.simpleExpressionAddingOperatorZeroOrMore =
simpleExpressionAddingOperatorZeroOrMore;
            this.line = line;
      }

      public String toString(){
            return "SimpleExpression[
"+simpleExpression_negZeroOrOne+","+term+","+simpleExpressionAddingOperatorZeroO
rMore+"]";
      }

            public Object visit(Visitor v, Object arg){
      return v.visitSimpleExpression(this, arg, line);
      }

}
```

mlast.SimpleExpressionAddingOperatorZeroOrMore.java

```
package mlast;
public class SimpleExpressionAddingOperatorZeroOrMore{
      public AddingOperator addingOperator;
      public Term term;
      public SimpleExpressionAddingOperatorZeroOrMore
simpleExpressionAddingOperatorZeroOrMore;
      public int line;
      public SimpleExpressionAddingOperatorZeroOrMore(AddingOperator
addingOperator, Term term, SimpleExpressionAddingOperatorZeroOrMore
simpleExpressionAddingOperatorZeroOrMore, int line){
            this.addingOperator = addingOperator;
            this.term = term;
            this.simpleExpressionAddingOperatorZeroOrMore =
simpleExpressionAddingOperatorZeroOrMore;
            this.line = line;
      }

      public String toString(){
            return "SimpleExpressionAddingOperatorZeroOrMore[
"+addingOperator+","+term+","+simpleExpressionAddingOperatorZeroOrMore+"]";
      }

            public Object visit(Visitor v, Object arg){
      return v.visitSimpleExpressionAddingOperatorZeroOrMore(this, arg, line);
      }

}
```

mlast.SimpleExpression_negZeroOrOne.java

```
package mlast;
public class SimpleExpression_negZeroOrOne{
      public String _negSpelling;
      public int line;

      public SimpleExpression_negZeroOrOne(String _negSpelling, int line){
            this._negSpelling = _negSpelling;
            this.line = line;
      }

      public String toString(){
            return "SimpleExpression_negZeroOrOne[ "+_negSpelling+"]";
      }

            public Object visit(Visitor v, Object arg){
      return v.visitSimpleExpression_negZeroOrOne(this, arg, line);
      }

}
```

mlast.SimpleType.java

```
package mlast;
public class SimpleType extends Ast{
public String spelling;
public int line;
public SimpleType(String Spelling, int line){
  this.spelling = Spelling;
  this.line = line;
}
public Object visit(Visitor v, Object arg){
  return v.visitSimpleType(this, arg, line);
}
}
```

mlast.StringLiteral.java

```
package mlast;
public class StringLiteral{
      public String spelling;
      public int line;

      public StringLiteral(String spelling, int line){
            this.spelling = spelling;
            this.line = line;
      }
```

```
	public String toString(){
		return "StringLiteral[ "+spelling+" ]";
	}

		public Object visit(Visitor v, Object arg){
	return v.visitStringLiteral(this, arg, line);
	}

}
```

mlast.Term.java

```
package mlast;
public class Term{
	public Factor factor;
	public TermMultiplyingOperatorZeroOrMore
termMultiplyingOperatorZeroOrMore;
	public int line;

	public Term(Factor factor, TermMultiplyingOperatorZeroOrMore
termMultiplyingOperatorZeroOrMore, int line){
		this.factor = factor;
		this.termMultiplyingOperatorZeroOrMore =
termMultiplyingOperatorZeroOrMore;
		this.line = line;
	}

	public String toString(){
		return "Term[ "+factor+","+termMultiplyingOperatorZeroOrMore+"]";
	}

		public Object visit(Visitor v, Object arg){
	return v.visitTerm(this, arg, line);
	}

}
```

mlast.TermMultiplyingOperatorZeroOrMore.java

```
package mlast;
public class TermMultiplyingOperatorZeroOrMore{
	public MultiplyingOperator multiplyingOperator;
	public Factor factor;
	public TermMultiplyingOperatorZeroOrMore
termMultiplyingOperatorZeroOrMore;
	public int line;

	public TermMultiplyingOperatorZeroOrMore(MultiplyingOperator
multiplyingOperator, Factor factor, TermMultiplyingOperatorZeroOrMore
termMultiplyingOperatorZeroOrMore, int line){
		this.multiplyingOperator = multiplyingOperator;
		this.factor = factor;
```

```
            this.termMultiplyingOperatorZeroOrMore =
termMultiplyingOperatorZeroOrMore;
            this.line = line;
      }

      public String toString(){
            return "TermMultiplyingOperatorZeroOrMore[
"+multiplyingOperator+","+factor+","+termMultiplyingOperatorZeroOrMore+"]";
      }

            public Object visit(Visitor v, Object arg){
      return v.visitTermMultiplyingOperatorZeroOrMore(this, arg, line);
      }

}
```

mlast.Type.java

```
package mlast;
public class Type{
      public SimpleType simpleType;
      public Type_listZeroOrOne type_listZeroOrOne;
      public int line;
      public Type(SimpleType simpleType, Type_listZeroOrOne type_listZeroOrOne,
int line){
            this.simpleType = simpleType;
            this.type_listZeroOrOne = type_listZeroOrOne;
            this.line = line;
      }

      public String toString(){
            return "Type[ "+simpleType+","+type_listZeroOrOne+"]";
      }

            public Object visit(Visitor v, Object arg){
      return v.visitType(this, arg, line);
      }

}
```

mlast.Type_listZeroOrOne.java

```
package mlast;
public class Type_listZeroOrOne{
      public String _listSpelling;
      public int line;

      public Type_listZeroOrOne(String _listSpelling, int line){
            this._listSpelling = _listSpelling;
            this.line = line;
      }
```

```
	public String toString(){
		return "Type_listZeroOrOne[ "+_listSpelling+"]";
	}

		public Object visit(Visitor v, Object arg){
	return v.visitType_listZeroOrOne(this, arg, line);
	}

}
```

mlast.Visitor.java

```
package mlast;
public interface Visitor{
	public Object visitProgram(Program program, Object arg, int line);
	public Object visitProgramFunctionsZeroOrMore(ProgramFunctionsZeroOrMore
programfunctionszeroormore, Object arg, int line);
	public Object visitCall(Call call, Object arg, int line);
	public Object visitFunctions(Functions functions, Object arg, int line);
	public Object visitIdentifier(Identifier identifier, Object arg, int
line);
	public Object visitFunctionsParListZeroOrOne(FunctionsParListZeroOrOne
functionsparlistzeroorone, Object arg, int line);
	public Object visitType(Type type, Object arg, int line);
	public Object visitExpression(Expression expression, Object arg, int
line);
	public Object visitParList(ParList parlist, Object arg, int line);
	public Object visitPar(Par par, Object arg, int line);
	public Object visitParList_commaZeroOrMore(ParList_commaZeroOrMore
parlist_commazeroormore, Object arg, int line);
	public Object visitSimpleType(SimpleType simpletype, Object arg, int
line);
	public Object visitType_listZeroOrOne(Type_listZeroOrOne
type_listzeroorone, Object arg, int line);
	public Object
visitExpressionConditionalExpression(ExpressionConditionalExpression
expressionconditionalexpression, Object arg, int line);
	public Object visitConditionalExpression(ConditionalExpression
conditionalexpression, Object arg, int line);
	public Object visitExpressionBasicExpression(ExpressionBasicExpression
expressionbasicexpression, Object arg, int line);
	public Object visitBasicExpression(BasicExpression basicexpression, Object
arg, int line);
	public Object visitPrimaryExpression(PrimaryExpression primaryexpression,
Object arg, int line);
	public Object
visitBasicExpressionPrimaryOperatorZeroOrMore(BasicExpressionPrimaryOperatorZero
OrMore basicexpressionprimaryoperatorzeroormore, Object arg, int line);
	public Object visitPrimaryOperator(PrimaryOperator primaryoperator, Object
arg, int line);
	public Object visitSimpleExpression(SimpleExpression simpleexpression,
Object arg, int line);
	public Object
visitPrimaryExpressionRelationalOperatorZeroOrOne(PrimaryExpressionRelationalOpe
```

```
ratorZeroOrOne primaryexpressionrelationaloperatorzeroorone, Object arg, int
line);
      public Object visitRelationalOperator(RelationalOperator
relationaloperator, Object arg, int line);
      public Object
visitSimpleExpression_negZeroOrOne(SimpleExpression_negZeroOrOne
simpleexpression_negzeroorone, Object arg, int line);
      public Object visitTerm(Term term, Object arg, int line);
      public Object
visitSimpleExpressionAddingOperatorZeroOrMore(SimpleExpressionAddingOperatorZero
OrMore simpleexpressionaddingoperatorzeroormore, Object arg, int line);
      public Object visitAddingOperator(AddingOperator addingoperator, Object
arg, int line);
      public Object visitFactor(Factor factor, Object arg, int line);
      public Object
visitTermMultiplyingOperatorZeroOrMore(TermMultiplyingOperatorZeroOrMore
termmultiplyingoperatorzeroormore, Object arg, int line);
      public Object visitMultiplyingOperator(MultiplyingOperator
multiplyingoperator, Object arg, int line);
      public Object visitFactor_integerliteral(Factor_integerliteral
factor_integerliteral, Object arg, int line);
      public Object visitIntegerLiteral(IntegerLiteral integerliteral, Object
arg, int line);
      public Object visitFactor_stringliteral(Factor_stringliteral
factor_stringliteral, Object arg, int line);
      public Object visitStringLiteral(StringLiteral stringliteral, Object arg,
int line);
      public Object visitFactor_identifier(Factor_identifier factor_identifier,
Object arg, int line);
      public Object
visitFactor_identifier_lparenZeroOrOne(Factor_identifier_lparenZeroOrOne
factor_identifier_lparenzeroorone, Object arg, int line);
      public Object
visitFactor_identifier_lparenZeroOrOneArgListZeroOrOne(Factor_identifier_lparenZ
eroOrOneArgListZeroOrOne factor_identifier_lparenzerooroneearglistzeroorone,
Object arg, int line);
      public Object visitArgList(ArgList arglist, Object arg, int line);
      public Object visitFactorListLiteral(FactorListLiteral factorlistliteral,
Object arg, int line);
      public Object visitListLiteral(ListLiteral listliteral, Object arg, int
line);
      public Object visitFactor_lparen(Factor_lparen factor_lparen, Object arg,
int line);
      public Object visitFactor_not(Factor_not factor_not, Object arg, int
line);
      public Object
visitListLiteralExpressionListZeroOrOne(ListLiteralExpressionListZeroOrOne
listliteralexpressionlistzeroorone, Object arg, int line);
      public Object visitExpressionList(ExpressionList expressionlist, Object
arg, int line);
      public Object
visitExpressionList_commaZeroOrMore(ExpressionList_commaZeroOrMore
expressionlist_commazeroormore, Object arg, int line);
      public Object visitCallArgListZeroOrOne(CallArgListZeroOrOne
callarglistzeroorone, Object arg, int line);
      public Object visitArgList_commaZeroOrMore(ArgList_commaZeroOrMore
arglist_commazeroormore, Object arg, int line);
```

```
}
```

mlparser.Error.java

```
package mlparser;
public class Error{
	public Error(String message, int line){
		System.out.println("Line " + line + ": " + message);
		System.exit(0);
	}

}
```

mlparser.Main.java

```
package mlparser;
import java.util.*;
public class Main{
	public static void main(String[] args){
		Parser parser = new Parser();
		mlast.Program program = parser.parse();
		mlast.Display display = new mlast.Display(program);
	}

}
```

mlparser.Parser.java

```
package mlparser;
import mlast.*;
public class Parser{
	private Token currentToken;
	Scanner scanner;

	private void accept(byte expectedKind){
		if(currentToken.getKind() == expectedKind)
			currentToken = (Token) scanner.scan();
		else
		new Error("Syntax error: " + currentToken.spelling + " is not
expected.  Expected "+expectedKind+" but got
"+currentToken.getKind()+".",currentToken.line);
	}

	private void acceptIt(){
		currentToken = (Token) scanner.scan();
	}

	public Program parse(){
		SourceFile sourceFile = new SourceFile();
		scanner = new Scanner(sourceFile.openFile());
```

```
            currentToken = (Token) scanner.scan();
            Program program = parseProgram();
            if(currentToken.getKind() != Token.EOT)
                  new Error("Syntax error: Redundant characters at the end of
program.",currentToken.line);
            return program;
            }

      public Identifier parseIdentifier(){
            Identifier identifier= new
Identifier(currentToken.spelling,currentToken.line);
            accept(Token.IDENTIFIER);
            return identifier;
      }

      public IntegerLiteral parseIntegerLiteral(){
            IntegerLiteral integerliteral= new
IntegerLiteral(currentToken.spelling,currentToken.line);
            accept(Token.INTEGERLITERAL);
            return integerliteral;
      }

      public StringLiteral parseStringLiteral(){
            StringLiteral stringliteral= new
StringLiteral(currentToken.spelling,currentToken.line);
            accept(Token.STRINGLITERAL);
            return stringliteral;
      }

//ProductionRule:ProductionRule[ NonTerminal[ Program ],Definition[
NonTerminalSymbol[ NonTerminal[ ProgramFunctionsZeroOrMore
]],null,SequenceDefinition[ NonTerminalSymbol[ NonTerminal[ Call
]],SequenceDefinition[ TerminalSymbol[ Terminal[ ";" ]],null]]]]
private Program parseProgram(){
//parse sequences
ProgramFunctionsZeroOrMore programFunctionsZeroOrMore = null;
if (currentToken.kind==Token.FUN){
programFunctionsZeroOrMore = parseProgramFunctionsZeroOrMore();
}
Call call = null;
call = parseCall();
String _semicolonSpelling = null;
_semicolonSpelling = currentToken.spelling;
accept(Token.SEMICOLON);
            return new
Program(programFunctionsZeroOrMore,call,_semicolonSpelling,currentToken.line);

}
//ProductionRule:ProductionRule[ NonTerminal[ ProgramFunctionsZeroOrMore
],Definition[ NonTerminalSymbol[ NonTerminal[ Functions
]],null,SequenceDefinition[ TerminalSymbol[ Terminal[ ";" ]],SequenceDefinition[
NonTerminalSymbol[ NonTerminal[ ProgramFunctionsZeroOrMore ]],null]]]]
private ProgramFunctionsZeroOrMore parseProgramFunctionsZeroOrMore(){
//parse sequences
Functions functions = null;
functions = parseFunctions();
String _semicolonSpelling = null;
```

```
_semicolonSpelling = currentToken.spelling;
accept(Token.SEMICOLON);
ProgramFunctionsZeroOrMore programFunctionsZeroOrMore = null;
if (currentToken.kind==Token.FUN){
programFunctionsZeroOrMore = parseProgramFunctionsZeroOrMore();
}
            return new
ProgramFunctionsZeroOrMore(functions,_semicolonSpelling,programFunctionsZeroOrMo
re,currentToken.line);

}
//ProductionRule:ProductionRule[ NonTerminal[ Functions ],Definition[
TerminalSymbol[ Terminal[ "fun" ]],null,SequenceDefinition[ NonTerminalSymbol[
NonTerminal[ Identifier ]],SequenceDefinition[ TerminalSymbol[ Terminal[ "("
]],SequenceDefinition[ NonTerminalSymbol[ NonTerminal[ FunctionsParListZeroOrOne
]],SequenceDefinition[ TerminalSymbol[ Terminal[ ")" ]],SequenceDefinition[
TerminalSymbol[ Terminal[ ":" ]],SequenceDefinition[ NonTerminalSymbol[
NonTerminal[ Type ]],SequenceDefinition[ TerminalSymbol[ Terminal[ "="
]],SequenceDefinition[ NonTerminalSymbol[ NonTerminal[ Expression
]],null]]]]]]]]]]
private Functions parseFunctions(){
//parse sequences
String _funSpelling = null;
_funSpelling = currentToken.spelling;
accept(Token.FUN);
Identifier identifier = null;
identifier = parseIdentifier();
String _lparenSpelling = null;
_lparenSpelling = currentToken.spelling;
accept(Token.LPAREN);
FunctionsParListZeroOrOne functionsParListZeroOrOne = null;
if (currentToken.kind==Token.IDENTIFIER){
functionsParListZeroOrOne = parseFunctionsParListZeroOrOne();
}
String _rparenSpelling = null;
_rparenSpelling = currentToken.spelling;
accept(Token.RPAREN);
String _colonSpelling = null;
_colonSpelling = currentToken.spelling;
accept(Token.COLON);
Type type = null;
type = parseType();
String _equalSpelling = null;
_equalSpelling = currentToken.spelling;
accept(Token.EQUAL);
Expression expression = null;
expression = parseExpression();
            return new
Functions(_funSpelling,identifier,_lparenSpelling,functionsParListZeroOrOne,_rpa
renSpelling,_colonSpelling,type,_equalSpelling,expression,currentToken.line);

}
//ProductionRule:ProductionRule[ NonTerminal[ FunctionsParListZeroOrOne
],Definition[ NonTerminalSymbol[ NonTerminal[ ParList ]],null,null]]
private FunctionsParListZeroOrOne parseFunctionsParListZeroOrOne(){
//parse sequences
ParList parList = null;
```

```
parList = parseParList();
            return new FunctionsParListZeroOrOne(parList,currentToken.line);

}
//ProductionRule:ProductionRule[ NonTerminal[ ParList ],Definition[
NonTerminalSymbol[ NonTerminal[ Par ]],null,SequenceDefinition[
NonTerminalSymbol[ NonTerminal[ ParList_commaZeroOrMore ]],null]]]
private ParList parseParList(){
//parse sequences
Par par = null;
par = parsePar();
ParList_commaZeroOrMore parList_commaZeroOrMore = null;
if (currentToken.kind==Token.COMMA){
parList_commaZeroOrMore = parseParList_commaZeroOrMore();
}
            return new ParList(par,parList_commaZeroOrMore,currentToken.line);

}
//ProductionRule:ProductionRule[ NonTerminal[ ParList_commaZeroOrMore
],Definition[ TerminalSymbol[ Terminal[ "," ]],null,SequenceDefinition[
NonTerminalSymbol[ NonTerminal[ Par ]],SequenceDefinition[ NonTerminalSymbol[
NonTerminal[ ParList_commaZeroOrMore ]],null]]]]
private ParList_commaZeroOrMore parseParList_commaZeroOrMore(){
//parse sequences
String _commaSpelling = null;
_commaSpelling = currentToken.spelling;
accept(Token.COMMA);
Par par = null;
par = parsePar();
ParList_commaZeroOrMore parList_commaZeroOrMore = null;
if (currentToken.kind==Token.COMMA){
parList_commaZeroOrMore = parseParList_commaZeroOrMore();
}
            return new
ParList_commaZeroOrMore(_commaSpelling,par,parList_commaZeroOrMore,currentToken.
line);

}
//ProductionRule:ProductionRule[ NonTerminal[ Par ],Definition[
NonTerminalSymbol[ NonTerminal[ Identifier ]],null,SequenceDefinition[
TerminalSymbol[ Terminal[ ":" ]],SequenceDefinition[ NonTerminalSymbol[
NonTerminal[ Type ]],null]]]]
private Par parsePar(){
//parse sequences
Identifier identifier = null;
identifier = parseIdentifier();
String _colonSpelling = null;
_colonSpelling = currentToken.spelling;
accept(Token.COLON);
Type type = null;
type = parseType();
            return new Par(identifier,_colonSpelling,type,currentToken.line);

}
//ProductionRule:ProductionRule[ NonTerminal[ Type ],Definition[
NonTerminalSymbol[ NonTerminal[ SimpleType ]],null,SequenceDefinition[
NonTerminalSymbol[ NonTerminal[ Type_listZeroOrOne ]],null]]]
```

```
private Type parseType(){
//parse sequences
SimpleType simpleType = null;
simpleType = parseSimpleType();
Type_listZeroOrOne type_listZeroOrOne = null;
if (currentToken.kind==Token.LIST){
type_listZeroOrOne = parseType_listZeroOrOne();
}
            return new Type(simpleType,type_listZeroOrOne,currentToken.line);


}
//ProductionRule:ProductionRule[ NonTerminal[ Type_listZeroOrOne ],Definition[
TerminalSymbol[ Terminal[ "list" ]],null,null]]
private Type_listZeroOrOne parseType_listZeroOrOne(){
//parse sequences
String _listSpelling = null;
_listSpelling = currentToken.spelling;
accept(Token.LIST);
            return new Type_listZeroOrOne(_listSpelling,currentToken.line);

}
//ProductionRule:ProductionRule[ NonTerminal[ SimpleType ],Definition[
TerminalSymbol[ Terminal[ "int" ]],AlternativeDefinition[ TerminalSymbol[
Terminal[ "real" ]],AlternativeDefinition[ TerminalSymbol[ Terminal[ "string"
]],null]],null]]
private SimpleType parseSimpleType(){
//Check Starter Sets
      switch (currentToken.kind){
            case Token.INT:
String _int = null;
_int = currentToken.spelling;
accept(Token.INT);
            return new SimpleType(_int,currentToken.line);
            case Token.REAL:
String _real = null;
_real = currentToken.spelling;
accept(Token.REAL);
            return new SimpleType(_real,currentToken.line);
            case Token.STRING:
String _string = null;
_string = currentToken.spelling;
accept(Token.STRING);
            return new SimpleType(_string,currentToken.line);
            default:
            new Error("Syntax error.  Expected _int or _real or _string.  Got
"+currentToken.getKind()+" spelled "+currentToken.getSpelling()+".",
currentToken.line);
      return null;
      }
}
//ProductionRule:ProductionRule[ NonTerminal[ Expression ],Definition[
NonTerminalSymbol[ NonTerminal[ ExpressionConditionalExpression
]],AlternativeDefinition[ NonTerminalSymbol[ NonTerminal[
ExpressionBasicExpression ]],null],null]]
private Expression parseExpression(){
//Check Starter Sets
```

```
      switch (currentToken.kind){
            case Token.IF:
ExpressionConditionalExpression ExpressionConditionalExpression = null;
ExpressionConditionalExpression = parseExpressionConditionalExpression();
            return ExpressionConditionalExpression;
            case Token.NOT:
            case Token.LBRACKET:
            case Token.LPAREN:
            case Token.NEG:
            case Token.STRINGLITERAL:
            case Token.INTEGERLITERAL:
            case Token.IDENTIFIER:
ExpressionBasicExpression ExpressionBasicExpression = null;
ExpressionBasicExpression = parseExpressionBasicExpression();
            return ExpressionBasicExpression;
            default:
            new Error("Syntax error.  Expected ExpressionConditionalExpression
or ExpressionBasicExpression.  Got "+currentToken.getKind()+" spelled
"+currentToken.getSpelling()+".", currentToken.line);
      return null;
      }
}
//ProductionRule:ProductionRule[ NonTerminal[ ExpressionConditionalExpression
],Definition[ NonTerminalSymbol[ NonTerminal[ ConditionalExpression
]],null,null]]
private ExpressionConditionalExpression parseExpressionConditionalExpression(){
//parse sequences
ConditionalExpression conditionalExpression = null;
conditionalExpression = parseConditionalExpression();
            return new
ExpressionConditionalExpression(conditionalExpression,currentToken.line);

}
//ProductionRule:ProductionRule[ NonTerminal[ ExpressionBasicExpression
],Definition[ NonTerminalSymbol[ NonTerminal[ BasicExpression ]],null,null]]
private ExpressionBasicExpression parseExpressionBasicExpression(){
//parse sequences
BasicExpression basicExpression = null;
basicExpression = parseBasicExpression();
            return new
ExpressionBasicExpression(basicExpression,currentToken.line);

}
//ProductionRule:ProductionRule[ NonTerminal[ ConditionalExpression
],Definition[ TerminalSymbol[ Terminal[ "if" ]],null,SequenceDefinition[
NonTerminalSymbol[ NonTerminal[ BasicExpression ]],SequenceDefinition[
TerminalSymbol[ Terminal[ "then" ]],SequenceDefinition[ NonTerminalSymbol[
NonTerminal[ Expression ]],SequenceDefinition[ TerminalSymbol[ Terminal[ "else"
]],SequenceDefinition[ NonTerminalSymbol[ NonTerminal[ Expression ]],null]]]]]]]
private ConditionalExpression parseConditionalExpression(){
//parse sequences
String _ifSpelling = null;
_ifSpelling = currentToken.spelling;
accept(Token.IF);
BasicExpression basicExpression = null;
basicExpression = parseBasicExpression();
String _thenSpelling = null;
```

```
_thenSpelling = currentToken.spelling;
accept(Token.THEN);
Expression expression = null;
expression = parseExpression();
String _elseSpelling = null;
_elseSpelling = currentToken.spelling;
accept(Token.ELSE);
Expression expression2 = null;
expression2 = parseExpression();
            return new
ConditionalExpression(_ifSpelling,basicExpression,_thenSpelling,expression,_else
Spelling,expression2,currentToken.line);

}
//ProductionRule:ProductionRule[ NonTerminal[ BasicExpression ],Definition[
NonTerminalSymbol[ NonTerminal[ PrimaryExpression ]],null,SequenceDefinition[
NonTerminalSymbol[ NonTerminal[ BasicExpressionPrimaryOperatorZeroOrMore
]],null]]]
private BasicExpression parseBasicExpression(){
//parse sequences
PrimaryExpression primaryExpression = null;
primaryExpression = parsePrimaryExpression();
BasicExpressionPrimaryOperatorZeroOrMore
basicExpressionPrimaryOperatorZeroOrMore = null;
if (currentToken.kind==Token.OR || currentToken.kind==Token.AND){
basicExpressionPrimaryOperatorZeroOrMore =
parseBasicExpressionPrimaryOperatorZeroOrMore();
}
            return new
BasicExpression(primaryExpression,basicExpressionPrimaryOperatorZeroOrMore,curre
ntToken.line);

}
//ProductionRule:ProductionRule[ NonTerminal[
BasicExpressionPrimaryOperatorZeroOrMore ],Definition[ NonTerminalSymbol[
NonTerminal[ PrimaryOperator ]],null,SequenceDefinition[ NonTerminalSymbol[
NonTerminal[ PrimaryExpression ]],SequenceDefinition[ NonTerminalSymbol[
NonTerminal[ BasicExpressionPrimaryOperatorZeroOrMore ]],null]]]]
private BasicExpressionPrimaryOperatorZeroOrMore
parseBasicExpressionPrimaryOperatorZeroOrMore(){
//parse sequences
PrimaryOperator primaryOperator = null;
primaryOperator = parsePrimaryOperator();
PrimaryExpression primaryExpression = null;
primaryExpression = parsePrimaryExpression();
BasicExpressionPrimaryOperatorZeroOrMore
basicExpressionPrimaryOperatorZeroOrMore = null;
if (currentToken.kind==Token.OR || currentToken.kind==Token.AND){
basicExpressionPrimaryOperatorZeroOrMore =
parseBasicExpressionPrimaryOperatorZeroOrMore();
}
            return new
BasicExpressionPrimaryOperatorZeroOrMore(primaryOperator,primaryExpression,basic
ExpressionPrimaryOperatorZeroOrMore,currentToken.line);

}
```

```
//ProductionRule:ProductionRule[ NonTerminal[ PrimaryOperator ],Definition[
TerminalSymbol[ Terminal[ "or" ]],AlternativeDefinition[ TerminalSymbol[
Terminal[ "and" ]],null],null]]
private PrimaryOperator parsePrimaryOperator(){
//Check Starter Sets
      switch (currentToken.kind){
            case Token.OR:
String _or = null;
_or = currentToken.spelling;
accept(Token.OR);
            return new PrimaryOperator(_or,currentToken.line);
            case Token.AND:
String _and = null;
_and = currentToken.spelling;
accept(Token.AND);
            return new PrimaryOperator(_and,currentToken.line);
            default:
            new Error("Syntax error.  Expected _or or _and.  Got
"+currentToken.getKind()+" spelled "+currentToken.getSpelling()+".",
currentToken.line);
      return null;
      }
}
//ProductionRule:ProductionRule[ NonTerminal[ PrimaryExpression ],Definition[
NonTerminalSymbol[ NonTerminal[ SimpleExpression ]],null,SequenceDefinition[
NonTerminalSymbol[ NonTerminal[ PrimaryExpressionRelationalOperatorZeroOrOne
]],null]]]
private PrimaryExpression parsePrimaryExpression(){
//parse sequences
SimpleExpression simpleExpression = null;
simpleExpression = parseSimpleExpression();
PrimaryExpressionRelationalOperatorZeroOrOne
primaryExpressionRelationalOperatorZeroOrOne = null;
if (currentToken.kind==Token.NEQ || currentToken.kind==Token.EQUAL ||
currentToken.kind==Token.GREATER || currentToken.kind==Token.LESS ||
currentToken.kind==Token.LEQ || currentToken.kind==Token.GEQ){
primaryExpressionRelationalOperatorZeroOrOne =
parsePrimaryExpressionRelationalOperatorZeroOrOne();
}
            return new
PrimaryExpression(simpleExpression,primaryExpressionRelationalOperatorZeroOrOne,
currentToken.line);

}
//ProductionRule:ProductionRule[ NonTerminal[
PrimaryExpressionRelationalOperatorZeroOrOne ],Definition[ NonTerminalSymbol[
NonTerminal[ RelationalOperator ]],null,SequenceDefinition[ NonTerminalSymbol[
NonTerminal[ SimpleExpression ]],null]]]
private PrimaryExpressionRelationalOperatorZeroOrOne
parsePrimaryExpressionRelationalOperatorZeroOrOne(){
//parse sequences
RelationalOperator relationalOperator = null;
relationalOperator = parseRelationalOperator();
SimpleExpression simpleExpression = null;
simpleExpression = parseSimpleExpression();
```

```
            return new
PrimaryExpressionRelationalOperatorZeroOrOne(relationalOperator,simpleExpression
,currentToken.line);

}
//ProductionRule:ProductionRule[ NonTerminal[ RelationalOperator ],Definition[
TerminalSymbol[ Terminal[ "<" ]],AlternativeDefinition[ TerminalSymbol[
Terminal[ "<=" ]],AlternativeDefinition[ TerminalSymbol[ Terminal[ ">"
]],AlternativeDefinition[ TerminalSymbol[ Terminal[ ">="
]],AlternativeDefinition[ TerminalSymbol[ Terminal[ "="
]],AlternativeDefinition[ TerminalSymbol[ Terminal[ "\=" ]],null]]]]],null]]
private RelationalOperator parseRelationalOperator(){
//Check Starter Sets
      switch (currentToken.kind){
            case Token.LESS:
String _less = null;
_less = currentToken.spelling;
accept(Token.LESS);
            return new RelationalOperator(_less,currentToken.line);
            case Token.LEQ:
String _leq = null;
_leq = currentToken.spelling;
accept(Token.LEQ);
            return new RelationalOperator(_leq,currentToken.line);
            case Token.GREATER:
String _greater = null;
_greater = currentToken.spelling;
accept(Token.GREATER);
            return new RelationalOperator(_greater,currentToken.line);
            case Token.GEQ:
String _geq = null;
_geq = currentToken.spelling;
accept(Token.GEQ);
            return new RelationalOperator(_geq,currentToken.line);
            case Token.EQUAL:
String _equal = null;
_equal = currentToken.spelling;
accept(Token.EQUAL);
            return new RelationalOperator(_equal,currentToken.line);
            case Token.NEQ:
String _neq = null;
_neq = currentToken.spelling;
accept(Token.NEQ);
            return new RelationalOperator(_neq,currentToken.line);
            default:
            new Error("Syntax error.  Expected _less or _leq or _greater or _geq
or _equal or _neq.  Got "+currentToken.getKind()+" spelled
"+currentToken.getSpelling()+".", currentToken.line);
      return null;
      }
}
//ProductionRule:ProductionRule[ NonTerminal[ SimpleExpression ],Definition[
NonTerminalSymbol[ NonTerminal[ SimpleExpression_negZeroOrOne
]],null,SequenceDefinition[ NonTerminalSymbol[ NonTerminal[ Term
]],SequenceDefinition[ NonTerminalSymbol[ NonTerminal[
SimpleExpressionAddingOperatorZeroOrMore ]],null]]]]
private SimpleExpression parseSimpleExpression(){
```

```
//parse sequences
SimpleExpression_negZeroOrOne simpleExpression_negZeroOrOne = null;
if (currentToken.kind==Token.NEG){
simpleExpression_negZeroOrOne = parseSimpleExpression_negZeroOrOne();
}
Term term = null;
term = parseTerm();
SimpleExpressionAddingOperatorZeroOrMore
simpleExpressionAddingOperatorZeroOrMore = null;
if (currentToken.kind==Token.MINUS || currentToken.kind==Token.PLUS){
simpleExpressionAddingOperatorZeroOrMore =
parseSimpleExpressionAddingOperatorZeroOrMore();
}
            return new
SimpleExpression(simpleExpression_negZeroOrOne,term,simpleExpressionAddingOperat
orZeroOrMore,currentToken.line);

}

//ProductionRule:ProductionRule[ NonTerminal[ SimpleExpression_negZeroOrOne
],Definition[ TerminalSymbol[ Terminal[ "~" ]],null,null]]
private SimpleExpression_negZeroOrOne parseSimpleExpression_negZeroOrOne(){
//parse sequences
String _negSpelling = null;
_negSpelling = currentToken.spelling;
accept(Token.NEG);
            return new
SimpleExpression_negZeroOrOne(_negSpelling,currentToken.line);

}
//ProductionRule:ProductionRule[ NonTerminal[
SimpleExpressionAddingOperatorZeroOrMore ],Definition[ NonTerminalSymbol[
NonTerminal[ AddingOperator ]],null,SequenceDefinition[ NonTerminalSymbol[
NonTerminal[ Term ]],SequenceDefinition[ NonTerminalSymbol[ NonTerminal[
SimpleExpressionAddingOperatorZeroOrMore ]],null]]]]
private SimpleExpressionAddingOperatorZeroOrMore
parseSimpleExpressionAddingOperatorZeroOrMore(){
//parse sequences
AddingOperator addingOperator = null;
addingOperator = parseAddingOperator();
Term term = null;
term = parseTerm();
SimpleExpressionAddingOperatorZeroOrMore
simpleExpressionAddingOperatorZeroOrMore = null;
if (currentToken.kind==Token.MINUS || currentToken.kind==Token.PLUS){
simpleExpressionAddingOperatorZeroOrMore =
parseSimpleExpressionAddingOperatorZeroOrMore();
}
            return new
SimpleExpressionAddingOperatorZeroOrMore(addingOperator,term,simpleExpressionAdd
ingOperatorZeroOrMore,currentToken.line);

}
//ProductionRule:ProductionRule[ NonTerminal[ AddingOperator ],Definition[
TerminalSymbol[ Terminal[ "+" ]],AlternativeDefinition[ TerminalSymbol[
Terminal[ "-" ]],null],null]]
private AddingOperator parseAddingOperator(){
```

```
//Check Starter Sets
      switch (currentToken.kind){
            case Token.PLUS:
String _plus = null;
_plus = currentToken.spelling;
accept(Token.PLUS);
            return new AddingOperator(_plus,currentToken.line);
            case Token.MINUS:
String _minus = null;
_minus = currentToken.spelling;
accept(Token.MINUS);
            return new AddingOperator(_minus,currentToken.line);
            default:
            new Error("Syntax error.  Expected _plus or _minus.  Got
"+currentToken.getKind()+" spelled "+currentToken.getSpelling()+".",
currentToken.line);
      return null;
      }
}
//ProductionRule:ProductionRule[ NonTerminal[ Term ],Definition[
NonTerminalSymbol[ NonTerminal[ Factor ]],null,SequenceDefinition[
NonTerminalSymbol[ NonTerminal[ TermMultiplyingOperatorZeroOrMore ]],null]]]
private Term parseTerm(){
//parse sequences
Factor factor = null;
factor = parseFactor();
TermMultiplyingOperatorZeroOrMore termMultiplyingOperatorZeroOrMore = null;
if (currentToken.kind==Token.ASTERISK || currentToken.kind==Token.INTDIV ||
currentToken.kind==Token.DIV || currentToken.kind==Token.MOD){
termMultiplyingOperatorZeroOrMore = parseTermMultiplyingOperatorZeroOrMore();
}
            return new
Term(factor,termMultiplyingOperatorZeroOrMore,currentToken.line);

}
//ProductionRule:ProductionRule[ NonTerminal[ TermMultiplyingOperatorZeroOrMore
],Definition[ NonTerminalSymbol[ NonTerminal[ MultiplyingOperator
]],null,SequenceDefinition[ NonTerminalSymbol[ NonTerminal[ Factor
]],SequenceDefinition[ NonTerminalSymbol[ NonTerminal[
TermMultiplyingOperatorZeroOrMore ]],null]]]]
private TermMultiplyingOperatorZeroOrMore
parseTermMultiplyingOperatorZeroOrMore(){
//parse sequences
MultiplyingOperator multiplyingOperator = null;
multiplyingOperator = parseMultiplyingOperator();
Factor factor = null;
factor = parseFactor();
TermMultiplyingOperatorZeroOrMore termMultiplyingOperatorZeroOrMore = null;
if (currentToken.kind==Token.ASTERISK || currentToken.kind==Token.INTDIV ||
currentToken.kind==Token.DIV || currentToken.kind==Token.MOD){
termMultiplyingOperatorZeroOrMore = parseTermMultiplyingOperatorZeroOrMore();
}
            return new
TermMultiplyingOperatorZeroOrMore(multiplyingOperator,factor,termMultiplyingOper
atorZeroOrMore,currentToken.line);

}
```

```
//ProductionRule:ProductionRule[ NonTerminal[ MultiplyingOperator ],Definition[
TerminalSymbol[ Terminal[ "*" ]],AlternativeDefinition[ TerminalSymbol[
Terminal[ "/" ]],AlternativeDefinition[ TerminalSymbol[ Terminal[ "div"
]],AlternativeDefinition[ TerminalSymbol[ Terminal[ "mod" ]],null]]],null]]
private MultiplyingOperator parseMultiplyingOperator(){
//Check Starter Sets
      switch (currentToken.kind){
            case Token.ASTERISK:
String _asterisk = null;
_asterisk = currentToken.spelling;
accept(Token.ASTERISK);
            return new MultiplyingOperator(_asterisk,currentToken.line);
            case Token.DIV:
String _div = null;
_div = currentToken.spelling;
accept(Token.DIV);
            return new MultiplyingOperator(_div,currentToken.line);
            case Token.INTDIV:
String _intdiv = null;
_intdiv = currentToken.spelling;
accept(Token.INTDIV);
            return new MultiplyingOperator(_intdiv,currentToken.line);
            case Token.MOD:
String _mod = null;
_mod = currentToken.spelling;
accept(Token.MOD);
            return new MultiplyingOperator(_mod,currentToken.line);

            default:
            new Error("Syntax error.  Expected _asterisk or _div or _intdiv or
_mod.  Got "+currentToken.getKind()+" spelled "+currentToken.getSpelling()+".",
currentToken.line);
      return null;
      }
}
//ProductionRule:ProductionRule[ NonTerminal[ Factor ],Definition[
NonTerminalSymbol[ NonTerminal[ Factor_integerliteral ]],AlternativeDefinition[
NonTerminalSymbol[ NonTerminal[ Factor_stringliteral ]],AlternativeDefinition[
NonTerminalSymbol[ NonTerminal[ Factor_identifier ]],AlternativeDefinition[
NonTerminalSymbol[ NonTerminal[ FactorListLiteral ]],AlternativeDefinition[
NonTerminalSymbol[ NonTerminal[ Factor_lparen ]],AlternativeDefinition[
NonTerminalSymbol[ NonTerminal[ Factor_not ]],null]]]]],null]]
private Factor parseFactor(){
//Check Starter Sets
      switch (currentToken.kind){
            case Token.INTEGERLITERAL:
Factor_integerliteral Factor_integerliteral = null;
Factor_integerliteral = parseFactor_integerliteral();
            return Factor_integerliteral;
            case Token.STRINGLITERAL:
Factor_stringliteral Factor_stringliteral = null;
Factor_stringliteral = parseFactor_stringliteral();
            return Factor_stringliteral;
            case Token.IDENTIFIER:
Factor_identifier Factor_identifier = null;
Factor_identifier = parseFactor_identifier();
            return Factor_identifier;
```

```
            case Token.LBRACKET:
FactorListLiteral FactorListLiteral = null;
FactorListLiteral = parseFactorListLiteral();
            return FactorListLiteral;
            case Token.LPAREN:
Factor_lparen Factor_lparen = null;
Factor_lparen = parseFactor_lparen();
            return Factor_lparen;
            case Token.NOT:
Factor_not Factor_not = null;
Factor_not = parseFactor_not();
            return Factor_not;
            default:
            new Error("Syntax error.  Expected Factor_integerliteral or
Factor_stringliteral or Factor_identifier or FactorListLiteral or Factor_lparen
or Factor_not.  Got "+currentToken.getKind()+" spelled
"+currentToken.getSpelling()+".", currentToken.line);
      return null;
      }
}
//ProductionRule:ProductionRule[ NonTerminal[ Factor_integerliteral
],Definition[ NonTerminalSymbol[ NonTerminal[ IntegerLiteral ]],null,null]]
private Factor_integerliteral parseFactor_integerliteral(){
//parse sequences
IntegerLiteral integerLiteral = null;
integerLiteral = parseIntegerLiteral();
            return new Factor_integerliteral(integerLiteral,currentToken.line);

}
//ProductionRule:ProductionRule[ NonTerminal[ Factor_stringliteral ],Definition[
NonTerminalSymbol[ NonTerminal[ StringLiteral ]],null,null]]
private Factor_stringliteral parseFactor_stringliteral(){
//parse sequences
StringLiteral stringLiteral = null;
stringLiteral = parseStringLiteral();
            return new Factor_stringliteral(stringLiteral,currentToken.line);

}
//ProductionRule:ProductionRule[ NonTerminal[ Factor_identifier ],Definition[
NonTerminalSymbol[ NonTerminal[ Identifier ]],null,SequenceDefinition[
NonTerminalSymbol[ NonTerminal[ Factor_identifier_lparenZeroOrOne ]],null]]]
private Factor_identifier parseFactor_identifier(){
//parse sequences
Identifier identifier = null;
identifier = parseIdentifier();
Factor_identifier_lparenZeroOrOne factor_identifier_lparenZeroOrOne = null;
if (currentToken.kind==Token.LPAREN){
factor_identifier_lparenZeroOrOne = parseFactor_identifier_lparenZeroOrOne();
}
            return new
Factor_identifier(identifier,factor_identifier_lparenZeroOrOne,currentToken.line
);

}
//ProductionRule:ProductionRule[ NonTerminal[ Factor_identifier_lparenZeroOrOne
],Definition[ TerminalSymbol[ Terminal[ "(" ]],null,SequenceDefinition[
NonTerminalSymbol[ NonTerminal[
```

```
Factor_identifier_lparenZeroOrOneArgListZeroOrOne ]],SequenceDefinition[
TerminalSymbol[ Terminal[ ")" ]],null]]]]
private Factor_identifier_lparenZeroOrOne
parseFactor_identifier_lparenZeroOrOne(){
//parse sequences
String _lparenSpelling = null;
_lparenSpelling = currentToken.spelling;
accept(Token.LPAREN);
Factor_identifier_lparenZeroOrOneArgListZeroOrOne
factor_identifier_lparenZeroOrOneArgListZeroOrOne = null;
if (currentToken.kind==Token.NOT || currentToken.kind==Token.LPAREN ||
currentToken.kind==Token.LBRACKET || currentToken.kind==Token.NEG ||
currentToken.kind==Token.STRINGLITERAL || currentToken.kind==Token.IF ||
currentToken.kind==Token.INTEGERLITERAL || currentToken.kind==Token.IDENTIFIER){
factor_identifier_lparenZeroOrOneArgListZeroOrOne =
parseFactor_identifier_lparenZeroOrOneArgListZeroOrOne();
}
String _rparenSpelling = null;
_rparenSpelling = currentToken.spelling;
accept(Token.RPAREN);
            return new
Factor_identifier_lparenZeroOrOne(_lparenSpelling,factor_identifier_lparenZeroOr
OneArgListZeroOrOne,_rparenSpelling,currentToken.line);

}
//ProductionRule:ProductionRule[ NonTerminal[
Factor_identifier_lparenZeroOrOneArgListZeroOrOne ],Definition[
NonTerminalSymbol[ NonTerminal[ ArgList ]],null,null]]
private Factor_identifier_lparenZeroOrOneArgListZeroOrOne
parseFactor_identifier_lparenZeroOrOneArgListZeroOrOne(){
//parse sequences
ArgList argList = null;
argList = parseArgList();
            return new
Factor_identifier_lparenZeroOrOneArgListZeroOrOne(argList,currentToken.line);

}
//ProductionRule:ProductionRule[ NonTerminal[ FactorListLiteral ],Definition[
NonTerminalSymbol[ NonTerminal[ ListLiteral ]],null,null]]
private FactorListLiteral parseFactorListLiteral(){
//parse sequences
ListLiteral listLiteral = null;
listLiteral = parseListLiteral();
            return new FactorListLiteral(listLiteral,currentToken.line);

}
//ProductionRule:ProductionRule[ NonTerminal[ Factor_lparen ],Definition[
TerminalSymbol[ Terminal[ "(" ]],null,SequenceDefinition[ NonTerminalSymbol[
NonTerminal[ Expression ]],SequenceDefinition[ TerminalSymbol[ Terminal[ ")"
]],null]]]]
private Factor_lparen parseFactor_lparen(){
//parse sequences
String _lparenSpelling = null;
_lparenSpelling = currentToken.spelling;
accept(Token.LPAREN);
Expression expression = null;
expression = parseExpression();
```

```
String _rparenSpelling = null;
_rparenSpelling = currentToken.spelling;
accept(Token.RPAREN);
            return new
Factor_lparen(_lparenSpelling,expression,_rparenSpelling,currentToken.line);

}
//ProductionRule:ProductionRule[ NonTerminal[ Factor_not ],Definition[
TerminalSymbol[ Terminal[ "not" ]],null,SequenceDefinition[ NonTerminalSymbol[
NonTerminal[ Factor ]],null]]]
private Factor_not parseFactor_not(){
//parse sequences
String _notSpelling = null;
_notSpelling = currentToken.spelling;
accept(Token.NOT);
Factor factor = null;
factor = parseFactor();
            return new Factor_not(_notSpelling,factor,currentToken.line);

}
//ProductionRule:ProductionRule[ NonTerminal[ ListLiteral ],Definition[
TerminalSymbol[ Terminal[ "[" ]],null,SequenceDefinition[ NonTerminalSymbol[
NonTerminal[ ListLiteralExpressionListZeroOrOne ]],SequenceDefinition[
TerminalSymbol[ Terminal[ "]" ]],null]]]]
private ListLiteral parseListLiteral(){
//parse sequences
String _lbracketSpelling = null;
_lbracketSpelling = currentToken.spelling;
accept(Token.LBRACKET);
ListLiteralExpressionListZeroOrOne listLiteralExpressionListZeroOrOne = null;
if (currentToken.kind==Token.NOT || currentToken.kind==Token.LPAREN ||
currentToken.kind==Token.LBRACKET || currentToken.kind==Token.NEG ||
currentToken.kind==Token.STRINGLITERAL || currentToken.kind==Token.IF ||
currentToken.kind==Token.INTEGERLITERAL || currentToken.kind==Token.IDENTIFIER){
listLiteralExpressionListZeroOrOne = parseListLiteralExpressionListZeroOrOne();
}
String _rbracketSpelling = null;
_rbracketSpelling = currentToken.spelling;
accept(Token.RBRACKET);
            return new
ListLiteral(_lbracketSpelling,listLiteralExpressionListZeroOrOne,_rbracketSpelli
ng,currentToken.line);

}
//ProductionRule:ProductionRule[ NonTerminal[ ListLiteralExpressionListZeroOrOne
],Definition[ NonTerminalSymbol[ NonTerminal[ ExpressionList ]],null,null]]
private ListLiteralExpressionListZeroOrOne
parseListLiteralExpressionListZeroOrOne(){
//parse sequences
ExpressionList expressionList = null;
expressionList = parseExpressionList();
            return new
ListLiteralExpressionListZeroOrOne(expressionList,currentToken.line);

}
```

```
//ProductionRule:ProductionRule[ NonTerminal[ ExpressionList ],Definition[
NonTerminalSymbol[ NonTerminal[ Expression ]],null,SequenceDefinition[
NonTerminalSymbol[ NonTerminal[ ExpressionList_commaZeroOrMore ]],null]]]
private ExpressionList parseExpressionList(){
//parse sequences
Expression expression = null;
expression = parseExpression();
ExpressionList_commaZeroOrMore expressionList_commaZeroOrMore = null;
if (currentToken.kind==Token.COMMA){
expressionList_commaZeroOrMore = parseExpressionList_commaZeroOrMore();
}
            return new
ExpressionList(expression,expressionList_commaZeroOrMore,currentToken.line);

}
//ProductionRule:ProductionRule[ NonTerminal[ ExpressionList_commaZeroOrMore
],Definition[ TerminalSymbol[ Terminal[ "," ]],null,SequenceDefinition[
NonTerminalSymbol[ NonTerminal[ Expression ]],SequenceDefinition[
NonTerminalSymbol[ NonTerminal[ ExpressionList_commaZeroOrMore ]],null]]]]
private ExpressionList_commaZeroOrMore parseExpressionList_commaZeroOrMore(){
//parse sequences
String _commaSpelling = null;
_commaSpelling = currentToken.spelling;
accept(Token.COMMA);
Expression expression = null;
expression = parseExpression();
ExpressionList_commaZeroOrMore expressionList_commaZeroOrMore = null;
if (currentToken.kind==Token.COMMA){
expressionList_commaZeroOrMore = parseExpressionList_commaZeroOrMore();
}
            return new
ExpressionList_commaZeroOrMore(_commaSpelling,expression,expressionList_commaZer
oOrMore,currentToken.line);

}
//ProductionRule:ProductionRule[ NonTerminal[ Call ],Definition[
NonTerminalSymbol[ NonTerminal[ Identifier ]],null,SequenceDefinition[
TerminalSymbol[ Terminal[ "(" ]],SequenceDefinition[ NonTerminalSymbol[
NonTerminal[ CallArgListZeroOrOne ]],SequenceDefinition[ TerminalSymbol[
Terminal[ ")" ]],null]]]]]
private Call parseCall(){
//parse sequences
Identifier identifier = null;
identifier = parseIdentifier();

String _lparenSpelling = null;
_lparenSpelling = currentToken.spelling;
accept(Token.LPAREN);
CallArgListZeroOrOne callArgListZeroOrOne = null;
if (currentToken.kind==Token.NOT || currentToken.kind==Token.LPAREN ||
currentToken.kind==Token.LBRACKET || currentToken.kind==Token.NEG ||
currentToken.kind==Token.STRINGLITERAL || currentToken.kind==Token.IF ||
currentToken.kind==Token.INTEGERLITERAL || currentToken.kind==Token.IDENTIFIER){
callArgListZeroOrOne = parseCallArgListZeroOrOne();
}
String _rparenSpelling = null;
_rparenSpelling = currentToken.spelling;
```

```
accept(Token.RPAREN);
            return new
Call(identifier,_lparenSpelling,callArgListZeroOrOne,_rparenSpelling,currentToke
n.line);

}
//ProductionRule:ProductionRule[ NonTerminal[ CallArgListZeroOrOne ],Definition[
NonTerminalSymbol[ NonTerminal[ ArgList ]],null,null]]
private CallArgListZeroOrOne parseCallArgListZeroOrOne(){
//parse sequences
ArgList argList = null;
argList = parseArgList();
            return new CallArgListZeroOrOne(argList,currentToken.line);

}
//ProductionRule:ProductionRule[ NonTerminal[ ArgList ],Definition[
NonTerminalSymbol[ NonTerminal[ Expression ]],null,SequenceDefinition[
NonTerminalSymbol[ NonTerminal[ ArgList_commaZeroOrMore ]],null]]]
private ArgList parseArgList(){
//parse sequences
Expression expression = null;
expression = parseExpression();
ArgList_commaZeroOrMore argList_commaZeroOrMore = null;
if (currentToken.kind==Token.COMMA){
argList_commaZeroOrMore = parseArgList_commaZeroOrMore();
}
            return new
ArgList(expression,argList_commaZeroOrMore,currentToken.line);

}
//ProductionRule:ProductionRule[ NonTerminal[ ArgList_commaZeroOrMore
],Definition[ TerminalSymbol[ Terminal[ "," ]],null,SequenceDefinition[
NonTerminalSymbol[ NonTerminal[ Expression ]],SequenceDefinition[
NonTerminalSymbol[ NonTerminal[ ArgList_commaZeroOrMore ]],null]]]]
private ArgList_commaZeroOrMore parseArgList_commaZeroOrMore(){
//parse sequences
String _commaSpelling = null;
_commaSpelling = currentToken.spelling;
accept(Token.COMMA);
Expression expression = null;
expression = parseExpression();
ArgList_commaZeroOrMore argList_commaZeroOrMore = null;
if (currentToken.kind==Token.COMMA){
argList_commaZeroOrMore = parseArgList_commaZeroOrMore();
}
            return new
ArgList_commaZeroOrMore(_commaSpelling,expression,argList_commaZeroOrMore,curren
tToken.line);

}

}
```

mlparser.Scanner.java

```
package mlparser;
import java.io.*;
import java.util.*;

import common.TokenI;
import common.Error;
import common.TokenRule;

public class Scanner{
      private List tokenRuleList = new ArrayList();

      private BufferedReader inFile;
      private char currentChar;
      //current tokenstring
      private StringBuffer currentTokenString;
      private static int line = 1;

      private List tokenList = new ArrayList();
      public Scanner(BufferedReader inFile){ List tokenRuleList = new
ArrayList();
      tokenRuleList.add(new TokenRule("SEPERATOR","\\x20|\\t",false,-2,true));
      tokenRuleList.add(new TokenRule("IDENTIFIER","[a-zA-Z]([a-zA-
Z]|\\d)*",false,0,false));
      tokenRuleList.add(new TokenRule("INTEGERLITERAL","\\d+",false,1,false));
      tokenRuleList.add(new TokenRule("STRINGLITERAL","'\\w'",false,2,false));
      tokenRuleList.add(new TokenRule("FUN","fun",true,3,false));
      tokenRuleList.add(new TokenRule("INT","int",true,4,false));
      tokenRuleList.add(new TokenRule("REAL","real",true,5,false));
      tokenRuleList.add(new TokenRule("STRING","string",true,6,false));
      tokenRuleList.add(new TokenRule("LIST","list",true,7,false));
      tokenRuleList.add(new TokenRule("ELSE","else",true,8,false));
      tokenRuleList.add(new TokenRule("IF","if",true,9,false));
      tokenRuleList.add(new TokenRule("THEN","then",true,10,false));
      tokenRuleList.add(new TokenRule("COLON",":",true,11,false));
      tokenRuleList.add(new TokenRule("SEMICOLON",";",true,12,false));
      tokenRuleList.add(new TokenRule("COMMA",",",true,13,false));
      tokenRuleList.add(new TokenRule("NEG","~",true,14,false));
      tokenRuleList.add(new TokenRule("NOT","not",true,15,false));
      tokenRuleList.add(new TokenRule("LPAREN","(",true,16,false));
      tokenRuleList.add(new TokenRule("RPAREN",")",true,17,false));
      tokenRuleList.add(new TokenRule("LBRACKET","[",true,18,false));
      tokenRuleList.add(new TokenRule("RBRACKET","]",true,19,false));
      tokenRuleList.add(new TokenRule("AND","and",true,20,false));
      tokenRuleList.add(new TokenRule("ASTERISK","*",true,21,false));
      tokenRuleList.add(new TokenRule("DIV","/",true,22,false));
      tokenRuleList.add(new TokenRule("MOD","mod",true,23,false));
      tokenRuleList.add(new TokenRule("INTDIV","div",true,24,false));
      tokenRuleList.add(new TokenRule("EQUAL","=",true,25,false));
      tokenRuleList.add(new TokenRule("GREATER",">",true,26,false));
      tokenRuleList.add(new TokenRule("GEQ",">=",true,27,false));
      tokenRuleList.add(new TokenRule("NEQ","\\=",true,28,false));
      tokenRuleList.add(new TokenRule("LESS","<",true,29,false));
      tokenRuleList.add(new TokenRule("LEQ","<=",true,30,false));
      tokenRuleList.add(new TokenRule("MINUS","-",true,31,false));
      tokenRuleList.add(new TokenRule("OR","or",true,32,false));
      tokenRuleList.add(new TokenRule("PLUS","+",true,33,false));
        this.inFile = inFile;
```

```
        this.tokenRuleList.addAll(tokenRuleList);
        try{
            int i = this.inFile.read();
            if(i == -1) //end of file
              currentChar = '\u0000';
            else
              currentChar = (char)i;
        }
        catch(IOException e){
              System.out.println(e);
        }
      }

      private void scanSeparator(){
        switch(currentChar){
            case '\n': case '\r':
              if(currentChar == '\n')
                  line++;
              discard();
        }
      }

      public TokenI scanAll(){
            currentTokenString = new StringBuffer("");
        while(currentChar == '\n' ||      currentChar == '\r')
            scanSeparator();

            if (tokenList.isEmpty()){
                  tokenList = scanToken();
            }

            TokenI token = (TokenI)tokenList.remove(0);
            return token;
      }

      public TokenI scan(){
            TokenI token = scanAll();
            while (token.getKind()<-1)
            token = scanAll();

      return token;
      }

      public List scanToken(){
            currentTokenString = new StringBuffer("");
            Token token = null;
        while(currentChar != '\n' && currentChar != '\r' && currentChar !=
'\u0000')
            takeIt();
            while (currentTokenString.length()!=0){
                  if (currentTokenString.toString().equals("\u0000")){
                        tokenList.add(new Token(TokenI.EOT,
currentTokenString.toString(), line));
                        return tokenList;
                  }

                  token = isMatch(currentTokenString.toString());
```

```
                  //System.out.println("Match:"+token);
                  currentTokenString =
currentTokenString.delete(0,token.spelling.length());
                  tokenList.add(token);
            }

            if (token==null)
                  tokenList.add( new Token(TokenI.EOT,
currentTokenString.toString(), line));
        //currentKind = scanToken();
        //return new Token(currentKind, currentSpelling.toString(), line);
        return tokenList;
      }

      private void discard(){
        try{
            int i = inFile.read();
            if(i == -1) //end of file
              currentChar = '\u0000';
            else
              currentChar = (char)i;
        }
        catch(IOException e){
              System.out.println(e);
        }
      }


      private void takeIt(){
            currentTokenString.append(currentChar);
        try{
            int i = inFile.read();
            if(i == -1) //end of file
              currentChar = '\u0000';
            else
              currentChar = (char)i;
        }
        catch(IOException e){
              System.out.println(e);
        }
      }

      private TokenI nextToken(){
            //currentTokenString
            return null;
      }

      /**
       * Loop through the rules and find the biggest match.
       * @param tokenString
       * @return
       */
      private Token isMatch(String tokenString){
            if (tokenString.length()==0){
                  new Error("wrong token [" + tokenString+"]", line);
                  return new Token(TokenI.EOT, tokenString, line);
            }
```

```
            for (int i=tokenRuleList.size()-1;i>-1;i--){
                  TokenRule tokenRule = (TokenRule) tokenRuleList.get(i);
                  if ( (tokenRule.isReservedWord() &&
tokenString.equals(tokenRule.getRegularExpression())) ||
(!tokenRule.isReservedWord() &&
tokenString.matches(tokenRule.getRegularExpression())) ){
                        return new Token(tokenRule.getKind(), tokenString,
line);
                  }
            }
            return isMatch(tokenString.substring(0,tokenString.length()-1));
      }


  public static void main(String[] args){


    SourceFile sourceFile = new SourceFile();
    TokenI token;
    Scanner s = new Scanner(sourceFile.openFile());
//http://java.sun.com/j2se/1.4.2/docs/api/java/util/regex/Pattern.html#sum
    do{
      token = s.scan();
      if (token.getKind()>=-1)
            System.out.println("Line: " + token.getLine() + ", spelling = [" +
token.getSpelling() + "], " + "kind = " + token.getKind());
    }while(token.getKind() != TokenI.EOT);
  }
}
```

mlparser.SourceFile.java

```
package mlparser;
import java.io.*;

public class SourceFile{
      public BufferedReader openFile() {
            BufferedReader inFile = null;
            inFile = new BufferedReader(new
InputStreamReader(ClassLoader.getSystemResourceAsStream("mlparser/test.txt")));
            return inFile;
      }

}
```

mlparser.Token.java

```
package mlparser;
import common.*;
import java.util.*;
```

```
public class Token implements TokenI{
public int kind;
public String spelling;
public int line;
public String toString(){
      return "Token line: "+line+" kind: "+kind+" spelling: "+spelling;
}

public Token(){
}

public Token(int kind, String spelling, int line){
this.kind = kind;
this.spelling = spelling;

this.line = line;
}

public String lookupConstantSpelling(String spelling){
      String result = (String) constantSpellingMap.get(spelling);
      return result;
}
public String lookupVariableSpelling(String spelling){
      return (String)variableSpellingMap.get(spelling);
}
public final static byte
      EOT = -1,
      SEPERATOR = -2,         //\x20|\t
      IDENTIFIER = 0,         //[a-zA-Z]([a-zA-Z]|\d)*
      INTEGERLITERAL = 1,           //\d+
      STRINGLITERAL = 2,            //'\w'
      FUN = 3,          //fun
      INT = 4,          //int
      REAL = 5,         //real
      STRING = 6,       //string
      LIST = 7,         //list
      ELSE = 8,         //else
      IF = 9,           //if
      THEN = 10,        //then
      COLON = 11,       //:
      SEMICOLON = 12,         //;
      COMMA = 13,       //,
      NEG = 14,         //~
      NOT = 15,         //not
      LPAREN = 16,            //(
      RPAREN = 17,            //)
      LBRACKET = 18,          //[
      RBRACKET = 19,          //]
      AND = 20,         //and
      ASTERISK = 21,          //*
      DIV = 22,         ///
      MOD = 23,         //mod
      INTDIV = 24,            //div
      EQUAL = 25,       //=
      GREATER = 26,           //>
      GEQ = 27,         //>=
      NEQ = 28,         //\=
```

```
	LESS = 29,		//<
	LEQ = 30,		//<=
	MINUS = 31,		//-
	OR = 32,		//or
	PLUS = 33;		//+
private final static Map constantSpellingMap = new HashMap();
static {
	constantSpellingMap.put("\\x20|\\t","SEPERATOR");
	constantSpellingMap.put("[a-zA-Z]([a-zA-Z]|\\d)*","IDENTIFIER");
	constantSpellingMap.put("\\d+","INTEGERLITERAL");
	constantSpellingMap.put("'\\w'","STRINGLITERAL");
	constantSpellingMap.put("fun","FUN");
	constantSpellingMap.put("int","INT");
	constantSpellingMap.put("real","REAL");
	constantSpellingMap.put("string","STRING");
	constantSpellingMap.put("list","LIST");
	constantSpellingMap.put("else","ELSE");
	constantSpellingMap.put("if","IF");
	constantSpellingMap.put("then","THEN");
	constantSpellingMap.put(":","COLON");
	constantSpellingMap.put(";","SEMICOLON");
	constantSpellingMap.put(",","COMMA");
	constantSpellingMap.put("~","NEG");
	constantSpellingMap.put("not","NOT");
	constantSpellingMap.put("(","LPAREN");
	constantSpellingMap.put(")","RPAREN");
	constantSpellingMap.put("[","LBRACKET");
	constantSpellingMap.put("]","RBRACKET");
	constantSpellingMap.put("and","AND");
	constantSpellingMap.put("*","ASTERISK");
	constantSpellingMap.put("/","DIV");
	constantSpellingMap.put("mod","MOD");
	constantSpellingMap.put("div","INTDIV");
	constantSpellingMap.put("=","EQUAL");
	constantSpellingMap.put(">","GREATER");
	constantSpellingMap.put(">=","GEQ");
	constantSpellingMap.put("\\=","NEQ");
	constantSpellingMap.put("<","LESS");
	constantSpellingMap.put("<=","LEQ");
	constantSpellingMap.put("-","MINUS");
	constantSpellingMap.put("or","OR");
	constantSpellingMap.put("+","PLUS");
};

private final static Map variableSpellingMap = new HashMap();
static {
	variableSpellingMap.put("SEPERATOR","\\x20|\\t");
	variableSpellingMap.put("IDENTIFIER","[a-zA-Z]([a-zA-Z]|\\d)*");
	variableSpellingMap.put("INTEGERLITERAL","\\d+");
	variableSpellingMap.put("STRINGLITERAL","'\\w'");
	variableSpellingMap.put("FUN","fun");
	variableSpellingMap.put("INT","int");
	variableSpellingMap.put("REAL","real");
	variableSpellingMap.put("STRING","string");
	variableSpellingMap.put("LIST","list");
	variableSpellingMap.put("ELSE","else");
	variableSpellingMap.put("IF","if");
```

```
        variableSpellingMap.put("THEN","then");
        variableSpellingMap.put("COLON",":");
        variableSpellingMap.put("SEMICOLON",";");
        variableSpellingMap.put("COMMA",",");
        variableSpellingMap.put("NEG","~");
        variableSpellingMap.put("NOT","not");
        variableSpellingMap.put("LPAREN","(");
        variableSpellingMap.put("RPAREN",")");
        variableSpellingMap.put("LBRACKET","[");
        variableSpellingMap.put("RBRACKET","]");
        variableSpellingMap.put("AND","and");
        variableSpellingMap.put("ASTERISK","*");
        variableSpellingMap.put("DIV","/");
        variableSpellingMap.put("MOD","mod");
        variableSpellingMap.put("INTDIV","div");
        variableSpellingMap.put("EQUAL","=");
        variableSpellingMap.put("GREATER",">");
        variableSpellingMap.put("GEQ",">=");
        variableSpellingMap.put("NEQ","\\=");
        variableSpellingMap.put("LESS","<");
        variableSpellingMap.put("LEQ","<=");
        variableSpellingMap.put("MINUS","-");
        variableSpellingMap.put("OR","or");
        variableSpellingMap.put("PLUS","+");
};

public int getKind() {
        return kind;
}

public int getLine() {
        return line;
}

public String getSpelling() {
        return spelling;
}

}
```

[i] Matt Davis
[ii] David A Watt pg 77
[iii] Andrew Cumming, "Functional languages are introduced"
[iv] David A Watt pg 7
[v] David A Watt pg 89-90
[vi] http://logging.apache.org/log4j/docs/index.html