Data Layouts

Data Structures For a Simple Compiler
Symbol Tables

Information about user defined names
Symbol Table

- Symbol Tables are organized for fast lookup.
  - Items are typically entered once and then looked up several times.
  - Hash Tables and Balanced Binary Search Trees are commonly used.
  - Each record contains a “name” (symbol) and information describing it.
Simple Hash Table

- Hasher translates “name” into an integer in a fixed range— the hash value.

- Hash Value indexes into an array of lists.
  - Entry with that symbol is in that list or is not stored at all.
  - Items with same hash value = bucket.
Self Organizing Hash Table

- Can achieve constant average time lookup if buckets have bounded average length.
- Can guarantee this if we periodically double number of hash buckets and re-hash all elements.
  » Can be done so as to minimize movement of items.
Self Organizing Hash Table

anObject

hasher

index

0

n

max

newhasher

n + max

2 * max

index

0

anObject
Balanced Binary Search Tree

- Binary search trees work if they are kept balanced.
- Can achieve logarithmic lookup time.
- Algorithms are somewhat complex.
  - Red-black trees and AVL trees are used.
  - No leaf is much farther from root than any other
Balanced Binary Search Tree
Symbol Tables + Blocks

- If a language is block structured then each block (scope) needs to be represented separately in the symbol table.
- If the hash table buckets are “stack-like” this is automatic.
- Can use a stack of balanced trees with one entry per scope.
Special Cases

- Some languages partition names into different classes—keywords, variable&function names, struct names...

- Separate symbol tables can then be used for each kind of name. The different symbol tables might have different characteristics.
  » hashtable-sortedlist-binarytree...
Parsing Information
Parse Trees

- The structure of a modern computer language is tree-like.
- Trees represent recursion well.
- A grammatical structure is a node with its parts as child nodes.
- Interior nodes are nonterminals.
- The tokens of the language are leaves.
Parse Trees

\[
\text{<statement>} ::= \text{<variable>} "::=" \text{<expression>}
\]

\[
x ::= a + 5
\]
Parse Trees

- There are different node types in the same tree.
- Variant records or type unions are typically used. Object-orientation is also useful here.
- Each node has a tag that distinguishes it, permitting testing on node type.
Parse Stack

- Parsing is often accomplished with a stack. (Not in this version of GCL)

- The stack holds values representing tokens, nonterminals and semantic symbols from the grammar.
  - It can either hold what is expected next (LL parsing) or what has already been seen (LR parsing)
Parse Stack

- A stack is used because most languages and their grammars are recursive. Stacks can accomplish much of what trees can.
- The contents of the stack are usually numeric encodings of the symbols for compactness of representation and speed of processing.
Parse Stack

Grammar fragment:

```plaintext
<statement> ::= <variable> "=" <expression> #doAssign
```

Example being scanned:

```plaintext
max := max + 1;
```
Stack vs Parameters

- In recursive descent parsing, no stack is needed.
- This is because the semantic records can be passed directly to the semantic routines as parameters.
- Semantic records can also be returned from the parsing functions.
Tokens

Information produced by the Scanner
Token Records

- Token records pass information about symbols scanned. This varies by token type.
- Variant records or type unions are typically used.
- Each value contains a tag - the token type - and additional information.
  » The tag is usually an integer.
Token Examples

- Simple tokens
- No additional info
- Only the tag field

- endNum

- Others are more complex
- Tag plus other info

- numeralNum
- 35
Handling Strings

- Strings are variable length and therefore present some problems.
- In C we can allocate a free-store object to hold the spelling--BUT, allocation is expensive in time.
- In Pascal, allocating fixed length strings is wasteful.
- Spell buffers are an alternative.
Strings in the Free Store

write “The answer is: “, x;

strval = new char[16];

The string is represented by the value of the pointer which can be passed around the compiler.
Strings in a Spell Buffer

write “The answer is: “, x;

before

The string is represented as (3,15) = (start, length)

after

The string is represented as (3,15) = (start, length)
Semantic Information
Semantic Information

- Parsing and semantic routines need to share information.
- This information can be passed as function parameters or a semantic stack can be used.
- There are different kinds of semantic information.
  - Variant Records/Type Unions/Objects
Semantic Records

- Each record needs a tag to distinguish its kind. We need to test the tag types.
- Depending on the tag there will be additional information.
- Sometimes the additional information must itself be a tagged union/variant record.
Simple Semantic Records

identifier
maximum
7

addoperator
+

reloperator
<=

ifentry
J35
J36
Complex Semantic Records

* see types (later)
In some compilers semantic records are stored in a semantic stack. In others, they are passed as parameters.
Type Information
Type Information

- Type information must be maintained for variables and parameters.
- There are different kinds of types
  - Variant Records/Type Unions/Objects
- There are different typing rules in different languages.
  - Pointers to records/structs are a simple representation.
Type Information

- Types describe variables.
  - size of a variable of this type (in bytes)
  - kind (tag)
  - additional information for some types.

- There are also recursive types.
Simple Types

integer 2

Boolean 2

character 1

The tag and the size are enough.
Tuple Type

[integer, Boolean]
Recursive Types

\[\text{[integer, [integer, Boolean]]}\]
Range Types

integer range[1..10]
Array Types

Boolean array[1..10][0..4]
Array Types (alternate)

Boolean array [range1] [range2]

```
array
  100
array
  10
Boolean
  2
range
  2
  1, 10
range
  2
  0, 4
integer
  2
integer
  2
```
Record Types

record [integer x, boolean y]

Note similarity to tuple types.
Pointer Types

pointer [integer, Boolean]
Procedure Types

proc (integer, Boolean)

Note: Not all languages have procedure types even when they have procedures.
Function Types

func (integer returns [integer, Boolean])

Note: Not all languages have function types even when they have functions.
Some languages (Java, Modula-3) permit a type to reference itself:

class node
{
    int value;
    node next;
}

The internal representation is a pointer (4 bytes)
Recursive Types Again

\[
\begin{align*}
\text{record} & \ [\text{integer array}[0..4] \ x, \ Boolean \ y] \ , \\
\text{integer range} & \ [1..10] \ , \\
\text{pointer} & \ [\text{integer}, \ \text{integer}] \ , \\
\text{func} & \ (\text{integer}, \ \text{Boolean} \ \text{returns} \ \text{integer array}[1..5]) \\
\end{align*}
\]

Left as an exercise. :-)}