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.
Simple Hash Table

*Diagram showing anObject and hasher connected to index with buckets.*
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

\[ \text{newhasher} = \text{index} \]

\[ \text{anObject} \]

\[ \text{hasher} \]

\[ \text{index} \]

\[ 0 \]

\[ n \]

\[ n + \max \]

\[ 2 \times \max \]
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

<statement> ::= <variable> “:=“ <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.
- 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

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

Example being scanned:

\[
\text{max := max + 1;}
\]
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

\begin{array}{c}
3 \quad N \quad a \quad m \quad e \\
\end{array}

\begin{array}{c}
\text{Name} \\
\end{array}

\begin{array}{c}
\text{The answer is:} \\
\end{array}

after

\begin{array}{c}
18 \quad N \quad a \quad m \quad e \quad T \quad h \quad e \quad a \quad n \quad s \quad w \quad e \quad r \quad i \quad s \quad i \quad s \\
\end{array}

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
- Add operator +
- Rel operator <=
- If entry J35 J36
Complex Semantic Records

- typeentry
- exprentry const 33
- integer 2
- exprentry variable 0, 6 false

* see types (later)
Semantic Stack

In some compilers semantic records are stored in a semantic stack.

```
<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>2</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Identifier</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>maximum</td>
<td>7</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>identifier</td>
<td>5</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>typeentry</td>
<td>2</td>
</tr>
</tbody>
</table>
```
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

The tag and the size are enough.
Tuple Type

\([\text{integer}, \text{Boolean}]\)
Recursive Types

[[integer, [integer, Boolean]]

tuple 6

integer 2
tuple 4

...
Range Types

integer range[1..10]
Array Types

Boolean array[1..10][0..4]

array
100
1, 10

array
10
0, 4

Boolean
2
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]
Pointer Types

pointer [integer, Boolean]
Procedure Types

proc (integer, Boolean)

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

```plaintext
func (integer returns [integer, Boolean])
```

Note: Not all languages have function types even when they have functions.
Self Recursive Types

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

[ record [integer array[0..4] x, Boolean y] ,
  integer range [1..10] ,
  pointer [integer, integer] ,
  func(integer, Boolean returns integer array[1..5])
]

Left as an exercise. :-}