chapter6: Introduction to Semantic Analysis; Syntax-Directed Translation

1. Introduction

- **Semantic Analysis Phase**
  - **Purpose**: compute additional information needed for compilation that is beyond the capabilities of Context-Free Grammars and Standard Parsing Algorithms
  - **It performs Static semantic analysis**: which takes place prior to execution
  - **The primarily tasks are:**
    - Building a symbol table, performing type inference and type checking.

- The static semantic analysis is described by
  - **Attribute grammar**
    - identify **attributes** of language entities that must be computed
    - and to write **attribute equations** or **semantic rules** that express how the computation of such attributes is related to the grammar rules of the language

- **Implementation** of the static semantic analysis:
  - Not as clearly expressible as parsing algorithms because of the addition problem caused by the timing of the analysis during the compilation process
  - Multi-pass (more common) or single pass lead to totally different process

2. Introduction to Type Checking

There is a level of correctness that goes beyond grammar and syntax.

example

```c
extern int foo(int a, int b, int c, int d);

int fee()
{
  int f[3], g[1], h, i, j, k;
  char *p;
  foo(h, i, "ab", j, k);
  k = f*i+j;
  h = g[17];
  printf("%s,%s\n", p, q);
  p = &(i+j);
  k = 3.14159;
  p = p*2;
}
```

Find the errors in the program.

- declared g[1], used g[17]
- wrong number of args to foo
- "ab" is not an int
- f declared as an array, used as a scalar
- 3.14159 is not an int
- & cannot be applied to a non-atomic expression
- * cannot be applied to pointers
None of these are simply syntax errors. They require a deeper analysis of the meaning (semantics) of the program.

3. Syntax Directed Definitions
The theoretical framework for semantic analysis uses *syntax directed definitions*, sometimes called *attribute grammars*. A syntax directed definition is an extension of a context free grammar in which each grammar symbol is assigned certain attributes.

Each production is augmented with semantic rules which are used to define the values of attributes. The attributes give meaning to the structures in the parse tree, such as:

- the data type of an expression or variable
- the value of a constant expression
- the number and types of parameters expected by a function
- the memory location and size of an expression
- the symbol table associated with a compound statement
- code generated to execute a statement or expression

The process of computing the attributes of each node is called *annotating* the parse tree, which is then called an *annotated* or *attributed* parse tree (or syntax tree).

One problem for compiler writers is that language references do not normally provide an attribute grammar, only the pure syntax of the context-free grammar. It is left to the compiler writer to construct an attribute grammar from the English language descriptions of the components of the programming language.

4. Attributes and Attribute Grammars
   - **Attributes**
     - Any property of a programming language construct such as:
       - The data type of a variable
       - The value of an expression
       - The location of a variable in memory
       - The object code of a procedure
       - The number of significant digits in a number
   - **Binding of the attribute**
     - The process of computing an attribute and associating its computed value with the language construct in question
   - **Binding time**
     - The time during the compilation/execution process when the binding of an attribute occurs
     - Based on the difference of the binding time, attributes is divided into *Static* attributes (be bound prior to execution) and *Dynamic* attributes (be bound during execution)
   - **Example:** The binding time and significance during compilation of the attributes.
     - Type checker
       - In a language like C or Pascal, is an important part of semantic analysis;
       - While in a language like LISP, data types are dynamic, LISP compiler must generate code to compute types and perform type checking during program execution.
     - The values of expressions
       - Usually dynamic and be computed during execution;
• But sometime can also be evaluated during compilation (constant folding).
  – The allocation of a variable:
    • Either static (such as in FORTRAN77) or dynamic (such as in LISP),
    • Sometimes it is a mixture of static and dynamic (such as in C and Pascal)
      depending on the language and properties of the variable itself.
  – Object code of a procedure:
    • A static attribute, which is computed by the code generator
  – Number of significant digits in a number:
    • Often not explicitly treated during compilation.

### 4.1 Attribute Grammars

• X.a means the value of ‘a’ associated to ‘X’
  – X is a grammar symbol and a is an attribute associated to X

• **Syntax-directed semantics:**
  – Attributes are associated directly with the grammar symbols of the language.
  – Given a collection of attributes a1, ..., ak, it implies that for each grammar rule
    X0 → X1X2...Xn (X0 is a nonterminal),
  – The values of the attributes Xi.aj of each grammar symbol Xi are related to the values
    of the attributes of the other symbols in the rule.

• An attribute grammar for attributes a1, a2, ..., ak is the collection of all attribute
equations or semantic rules of the following form,
  – for all the grammar rules of the language.
    • Xi.aj = fij(X0.a1,...,X0.ak, ..., X1.al, ..., Xn-1.a1, ...Xn.ak)
    • Where fij is a mathematical function of its arguments

• Typically, attribute grammars are written in tabular form as follows:

<table>
<thead>
<tr>
<th>Grammar Rule</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number1 → number2 digit</td>
<td>number1.val = number2.val * 10 + digit.val</td>
</tr>
<tr>
<td>Number → digit</td>
<td>number.val = digit.val</td>
</tr>
<tr>
<td>digit → 0</td>
<td>digit.val = 0</td>
</tr>
<tr>
<td>digit → 1</td>
<td>digit.val = 1</td>
</tr>
<tr>
<td>digit → 2</td>
<td>digit.val = 2</td>
</tr>
<tr>
<td>digit → 3</td>
<td>digit.val = 3</td>
</tr>
<tr>
<td>digit → 4</td>
<td>digit.val = 4</td>
</tr>
<tr>
<td>digit → 5</td>
<td>digit.val = 5</td>
</tr>
<tr>
<td>digit → 6</td>
<td>digit.val = 6</td>
</tr>
<tr>
<td>digit → 7</td>
<td>digit.val = 7</td>
</tr>
<tr>
<td>digit → 8</td>
<td>digit.val = 8</td>
</tr>
<tr>
<td>digit → 9</td>
<td>digit.val = 9</td>
</tr>
</tbody>
</table>

| Table 6.1 |
The parse tree showing attribute computations for the number 345 is given as follows:

![Parse Tree Diagram]

**Example 6.2** consider the following grammar for simple integer arithmetic expressions:

- $\text{Exp} \rightarrow \text{exp} + \text{term} | \text{exp-term} | \text{term}$
- $\text{Term} \rightarrow \text{term*factor} | \text{factor}$
- $\text{Factor} \rightarrow (\text{exp}) | \text{number}$

The principal attribute of an exp (or term or factor) is its numeric value (write as `val`) and the attribute equations for the `val` attribute are given as follows:

<table>
<thead>
<tr>
<th>Grammar Rule</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{exp1} \rightarrow \text{exp2} + \text{term}$</td>
<td>$\text{exp1.val} = \text{exp2.val} + \text{term.val}$</td>
</tr>
<tr>
<td>$\text{exp1} \rightarrow \text{exp2} - \text{term}$</td>
<td>$\text{exp1.val} = \text{exp2.val} - \text{term.val}$</td>
</tr>
<tr>
<td>$\text{exp1} \rightarrow \text{term}$</td>
<td>$\text{exp1.val} = \text{term.val}$</td>
</tr>
<tr>
<td>$\text{term1} \rightarrow \text{term2*factor}$</td>
<td>$\text{term1.val} = \text{term2.val*factor.val}$</td>
</tr>
<tr>
<td>$\text{term} \rightarrow \text{factor}$</td>
<td>$\text{term.val} = \text{factor.val}$</td>
</tr>
<tr>
<td>$\text{factor} \rightarrow (\text{exp})$</td>
<td>$\text{factor.val} = \text{exp.val}$</td>
</tr>
<tr>
<td>$\text{factor} \rightarrow \text{number}$</td>
<td>$\text{factor.val} = \text{number.val}$</td>
</tr>
</tbody>
</table>

Given the expression $(34-3)*42$, the computations implied by this attribute grammar by attaching equations to nodes in a parse tree is as follows:

![Parse Tree Diagram]
Example 6.3 consider the following simple grammar of variable declarations in a C-like syntax:

\[
\begin{align*}
\text{Decl} & \rightarrow \text{type} \ \text{var-list} \\
\text{type} & \rightarrow \text{int} \mid \text{float} \\
\text{Var-list} & \rightarrow \text{id}, \ \text{var-list} \mid \text{id}
\end{align*}
\]

Define a data type attribute for the variables given by the identifiers in a declaration and write equations expressing how the data type attribute is related to the type of the declaration as follows:

(We use the name dtype to distinguish the attribute from the nonterminal type)

<table>
<thead>
<tr>
<th>Grammar Rule</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>decl \rightarrow \text{type} \ \text{var-list}</td>
<td>var-list.dtype = type.dtype</td>
</tr>
<tr>
<td>type \rightarrow \text{int}</td>
<td>type.dtype = integer</td>
</tr>
<tr>
<td>type \rightarrow \text{float}</td>
<td>type.dtype = real</td>
</tr>
<tr>
<td>var-list1 \rightarrow \text{id}, \text{var-list2}</td>
<td>id.dtype = var-list1.dtype</td>
</tr>
<tr>
<td>\text{var-list2}.dtype = var-list1.dtype</td>
<td></td>
</tr>
<tr>
<td>var-list \rightarrow \text{id}</td>
<td>id.type = var-list.dtype</td>
</tr>
</tbody>
</table>

Table 6.3

Note that there is no equation involving the dtype of the nonterminal decl.

It is not necessary for the value of an attribute to be specified for all grammar symbols

Parse tree for the string \texttt{float x,y} showing the dtype attribute as specified by the attribute grammar above is as follows:

![Parse tree](image)

Attribute grammars may involve several interdependent attributes.
Example 6.4 consider the following grammar, where numbers may be octal or decimal, suppose this is indicated by a one-character suffix o(for octal) or d(for decimal):

Based-num → num basechar
Basechar → o|d
Num → num digit | digit
Digit → 0|1|2|3|4|5|6|7|8|9

In this case num and digit require a new attribute base, which is used to compute the val attribute. The attribute grammar for base and val is given as follows.

<table>
<thead>
<tr>
<th>Grammar Rule</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Based-num → num basechar</td>
<td>Based-num.val = num.val</td>
</tr>
<tr>
<td>Basechar → o</td>
<td>Basechar.base = 8</td>
</tr>
<tr>
<td>Basechar → d</td>
<td>Basechar.base = 10</td>
</tr>
<tr>
<td>Num1 → num2 digit</td>
<td>num1.val =</td>
</tr>
<tr>
<td></td>
<td>If digit.val = error or num2.val = error</td>
</tr>
<tr>
<td></td>
<td>Then error</td>
</tr>
<tr>
<td></td>
<td>Else num2.val*num1.base + digit.val</td>
</tr>
<tr>
<td>Num → digit</td>
<td>Num2.base = num1.base</td>
</tr>
<tr>
<td>Digit → 0</td>
<td>Dig.base = num1.base</td>
</tr>
<tr>
<td>Digit → 1</td>
<td>digit.val = 0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Digit → 7</td>
<td>digit.val = 7</td>
</tr>
<tr>
<td>Digit → 8</td>
<td>digit.val = if digit.base = 8 then error else 8</td>
</tr>
<tr>
<td>Digit → 9</td>
<td>digit.val = if digit.base = 8 then error else 9</td>
</tr>
</tbody>
</table>

Tab 6.4
Two new features should be noted in this attribute grammar. First, the BNF grammar does not itself eliminate the erroneous combination of the (non-octal) digits 8 and 9 with the \texttt{o} suffix. For instance, the string 189o is syntactically correct according to the above BNF, but cannot be assigned any value. Thus, a new \texttt{error} value is needed for such cases. Additionally, the attribute grammar must express the fact that the inclusion of 8 or 9 in a number with an \texttt{o} suffix results in an \texttt{error} value. The easiest way to do this is to use an \texttt{if-then-else} expression in the functions of the appropriate attribute equations. For instance, the equation

\[
\text{num}_1 . \text{val} = \begin{cases} 
\text{error} & \text{if digit.val = error or num}_2 . \text{val} = \text{error} \\
\text{error} & \text{then error} \\
\text{else num}_2 . \text{val} * \text{num}_1 . \text{base} + \text{digit.val}
\end{cases}
\]

corresponding to the grammar rule \(\text{num}_1 \rightarrow \text{num}_2 \text{ digit}\) expresses the fact that if either of \(\text{num}_2 . \text{val}\) or \(\text{digit.val}\) are \texttt{error} then \(\text{num}_1 . \text{val}\) must also be \texttt{error}, and only if that is not the case is \(\text{num}_1 . \text{val}\) given by the formula \(\text{num}_2 . \text{val} \times \text{num}_1 . \text{base} + \text{digit.val}\).

To conclude this example, we again show the attribute calculations on a parse tree. Figure 6.4 gives a parse tree for the number 345o, together with the attribute values computed according to the attribute grammar of Table 6.4.

![Parse Tree for Number 345o](image-url)