Semantic Analysis

- Parser verifies that a program is syntactically correct and constructs a syntax tree (or other intermediate representation).
- Semantic analyzer checks that the program satisfies all other static language requirements (is “meaningful”) and collects and computes information needed for code generation.
Semantic Analysis Tasks

• Have variables been declared before use?
• Have variables been declared twice in the same scope?
• Has every declared variable been used?
• Are the variable and expression in an assignment type-compatible?
• Do the operands of (arithmetic) operators have compatible types?
• Do the arguments in a function call match the parameters of the function definition in number and type?
• Are arguments passed by reference variables?
Important Semantic Information

- Symbol table: collects declaration and scope information to satisfy “declaration before use” rule, and to establish data type and other properties of names in a program.

- Data types and type checking: compute data types for all typed language entities and check that language rules on types are satisfied.
How to build the symbol table and check types:

- Analyze the scope rules for the language and determine an appropriate table structure for maintaining this information.
- Analyze the type requirements and translate them into rules that can be applied recursively on a syntax tree.
Theoretical framework for semantic analysis

- Focus on attributes: computable properties of language constructs that are needed to satisfy language requirements and/or generate code
- Describe the computation of attributes using equations or algorithms.
- Associate these equations to grammar rules and/or kinds of nodes in a syntax tree.
• Analyze the structure of the equations to determine an order in which the attributes can be computed. (Tree traversals of syntax tree - preorder, postorder, inorder, or some combination of them.)
• Such a set of equations, functions and conditions is called an attribute grammar.
• Formally describing the evaluation of attributes and the conditions that attributes must satisfy using an attribute grammar helps significantly, even if attribute grammar tools are not used for semantic analysis.
• Tools such as GAG and Eli generate semantic analysers from attribute grammar specifications.
• Tools such as Yacc implicitly use attribute grammars in their semantic actions.
Example of an attribute grammar

Grammar:

\[
\begin{align*}
\text{exp} & \rightarrow \text{exp} + \text{term} \mid \text{exp} - \text{term} \mid \text{term} \\
\text{term} & \rightarrow \text{term} \ast \text{factor} \mid \text{factor} \\
\text{factor} & \rightarrow (\text{exp}) \mid \text{number}
\end{align*}
\]

Attribute Grammar:

<table>
<thead>
<tr>
<th>GRAMMAR RULE</th>
<th>SEMANTIC RULES</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{exp}_1 \rightarrow \text{exp}_2 + \text{term} )</td>
<td>( \text{exp}_1.\text{val} = \text{exp}_2.\text{val} + \text{term}.\text{val} )</td>
</tr>
<tr>
<td>( \text{exp}_1 \rightarrow \text{exp}_2 - \text{term} )</td>
<td>( \text{exp}_1.\text{val} = \text{exp}_2.\text{val} - \text{term}.\text{val} )</td>
</tr>
<tr>
<td>( \text{exp} \rightarrow \text{term} )</td>
<td>( \text{exp}.\text{val} = \text{term}.\text{val} )</td>
</tr>
<tr>
<td>( \text{term}_1 \rightarrow \text{term}_2 \ast \text{factor} )</td>
<td>( \text{term}_1.\text{val} = \text{term}_2.\text{val} \ast \text{factor}.\text{val} )</td>
</tr>
<tr>
<td>( \text{term} \rightarrow \text{factor} )</td>
<td>( \text{term}.\text{val} = \text{factor}.\text{val} )</td>
</tr>
<tr>
<td>( \text{factor} \rightarrow (\text{exp}) )</td>
<td>( \text{factor}.\text{val} = \text{exp}.\text{val} )</td>
</tr>
<tr>
<td>( \text{factor} \rightarrow \text{number} )</td>
<td>( \text{factor}.\text{val} = \text{number}.\text{val} )</td>
</tr>
</tbody>
</table>
Notes:

- Different instances of same nonterminal must be subscripted to distinguish them.
- Some attributes must have been precomputed (by scanner or parser), e.g. `number.val`.
- These particular attribute equations look a lot like a yacc specification, because they represent a bottom-up attribute computation.
A Second Example

Grammar:

\[
decl \rightarrow \text{type var-list} \\
\text{type} \rightarrow \text{int} \mid \text{float} \\
\text{var-list} \rightarrow \text{id, var-list} \mid \text{id}
\]

Attribute Grammar:

<table>
<thead>
<tr>
<th>Grammar Rule</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>( decl \rightarrow \text{type var-list} )</td>
<td>( \text{var-list}.\text{dtype} = \text{type}.\text{dtype} )</td>
</tr>
<tr>
<td>( \text{type} \rightarrow \text{int} )</td>
<td>( \text{type}.\text{dtype} = \text{integer} )</td>
</tr>
<tr>
<td>( \text{type} \rightarrow \text{float} )</td>
<td>( \text{type}.\text{dtype} = \text{real} )</td>
</tr>
<tr>
<td>( \text{var-list}_1 \rightarrow \text{id, var-list}_2 )</td>
<td>( \text{id}.\text{dtype} = \text{var-list}_1.\text{dtype} )</td>
</tr>
<tr>
<td>( \text{var-list} \rightarrow \text{id} )</td>
<td>( \text{var-list}_2.\text{dtype} = \text{var-list}_1.\text{dtype} )</td>
</tr>
<tr>
<td></td>
<td>( \text{id}.\text{dtype} = \text{var-list}.\text{dtype} )</td>
</tr>
</tbody>
</table>
Notes

- Data type typically propagates *down* a syntax tree via declarations.
- No longer something yacc can handle directly.
- Such an attribute is called *inherited*, while bottom-up calculation is called *synthesized*.
- Syntax tree is a standard synthesized attribute computable by yacc; other attributes computed on the tree.
Dependency graph

- Indicates order in which attributes must be computed.
- Synthesized attributes always flow from children to parents, and can always be computed by a postorder traversal.
- Inherited attributes can flow any other way.
- *L-attributed*: a left-to-right traversal suffices to compute attributes. However, this may involve a combination of preorder, inorder, and postorder traversal.
Data type dependencies (by grammar rule):

\[\text{decl} \rightarrow \text{type var-list}:\]
\[\text{var-list} . \text{dtype} = \text{type} . \text{dtype}\]

\[\text{var-list} \rightarrow \text{id}, \text{var-list}:\]
\[\text{id} . \text{dtype} = \text{var-list}_{1} . \text{dtype}\]
\[\text{var-list}_{2} . \text{dtype} = \text{var-list}_{1} . \text{dtype}\]
L-attributed dependencies have three basic mechanisms:

(a) Inheritance from parent to siblings

(b) Inheritance from sibling to sibling via the parent

(c) Sibling inheritance via sibling pointers
Sample tree structure:

typedef enum {decl, type, id} nodekind;
typedef enum {integer, real} typekind;
typedef struct treeNode
    { nodekind kind;
      struct treeNode
         * lchild, * rchild, * sibling;
      typekind dtype;
      /\* for type and id nodes */
      char * name;
      /\* for id nodes only */
    } * SyntaxTree;
Sample tree instance:

String: float x, y

Tree:

```
  decl
  /   \
type ( dtype = real )
   /   \
  id ( x )     id ( y )
```
Traversals code:

```c
void evalType (SyntaxTree t)
{
    switch (t->kind)
    {
    case decl:
        t->rchild->dtype = t->lchild->dtype;
        evalType(t->rchild);
        break;
    case id:
        if (t->sibling != NULL)
            { t->sibling->dtype = t->dtype;
              evalType(t->sibling);
            }
        break;
    } /* end switch */
} /* end evalType */
```
Attributes need not be kept in the syntax tree:

<table>
<thead>
<tr>
<th>Grammar Rule</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>decl → type var-list</td>
<td></td>
</tr>
<tr>
<td>type → int</td>
<td>dtype = integer</td>
</tr>
<tr>
<td>type → float</td>
<td>dtype = real</td>
</tr>
<tr>
<td>var-list₁ → id , var-list₂</td>
<td>insert(id .name, dtype)</td>
</tr>
<tr>
<td>var-list → id</td>
<td>insert(id .name, dtype)</td>
</tr>
</tbody>
</table>

**dtype** is global

Use a symbol table to store the type of each identifier
New traversal code:

typekind dtype; /* global */
void evalType (SyntaxTree t)
{ switch (t->kind)
    { case decl:
            dtype = t->lchild->dtype;
            evalType(t->rchild);
            break;
        case id:
            insert(t->name,dtype);
            if (t->sibling != NULL)
                evalType(t->sibling);
            break;
    } /* end switch */
} /* end evalType */
Even better, use a parameter instead of a global variable:

```c
void evalDecl(SyntaxTree t)
{
    evalType(t->rchild, t->lchild->dtype);
}

void evalType(SyntaxTree t, typekind dtype)
{
    insert(t->name, dtype);
    if (t->sibling != NULL)
        evalType(t->sibling, dtype);
}
```

Note: inherited attributes can often be turned into parameters to recursive traversal functions, while synthesized attributes can be turned into returned values.
Alternative to a difficult inherited situation (not recommended):

**Theorem** (Knuth [1968]). Given an attribute grammar, all inherited attributes can be changed into synthesized attributes by suitable modification of the grammar, without changing the language of the grammar.
Example:

New grammar for types:
\[\text{decl} \rightarrow \text{var-list} \ \text{id}\]
\[\text{var-list} \rightarrow \text{var-list} \ \text{id}, \ | \ \text{type}\]
\[\text{type} \rightarrow \text{int} \ | \ \text{float}\]

New Tree for float \(x, y\) might be:

\[
\text{id} \ (y) \\
(\text{dtype} = \text{real})
\]
\[
\text{id} \ (x) \\
(\text{dtype} = \text{real})
\]
\[
\text{type} \\
(\text{dtype} = \text{real})
\]
Our approach:

- Compute inherited stuff first (symbol table) in a separate pass
- Then type inference and type checking turns into a purely synthesized attribute computation, since all uses of names have their types already computed.

Next:

- Symbol table structure
- Synthesized type rules