Semantic Processing

- The Semantic Processing phase consists of:
  - Checking the Static Semantics of the language
  - Generating an Intermediate Representation of the program
- Checking the static semantics include:
  - Making sure that all identifiers used in a program are declared
  - Making sure that all functions called are declared or defined
  - Making sure that parameters are passed correctly
  - Checking the uses of operators and types of expressions
  - Entering identifiers in symbol tables
Attribute Grammars

- Provides a practical formalism for describing semantic processing
- Proposed by Knuth in 1968
- Each grammar symbol has an associated set of attributes
- An attribute can represent anything we choose
  - The value of an expression when literal constants are used
  - The data type of a constant, variable, or expression
  - The location (or offset) of a variable in memory
  - The translated code of an expression, statement, or function
- An annotated or attributed parse tree is a
  - Parse tree showing the values of attributes at each node
- Attributes may be evaluated on the fly as an input is parsed
- Alternatively, attributes may be also evaluated after parsing
Synthesized and Inherited Attributes

- The attributes are divided into two classes:
  - Synthesized Attributes
  - Inherited Attributes

- A synthesized attribute of a parse tree node is computed from
  - Attribute values of the children nodes

- An inherited attribute of a parse tree node is computed from
  - Attribute values of the parent node
  - Attribute values of the sibling nodes

- Tokens may have only synthesized attributes
  - Token attributes are supplied by the scanner

- Nonterminals may have synthesized and/or inherited attributes

- Attributes are evaluated according to Semantic rules
  - Semantic rules are associated with production rules
S-Attributed Grammars

- S-Attributed grammars allow only synthesized attributes
- Synthesized attributes are evaluated bottom up
- S-Attributed grammars work perfectly with LR parsers
- Consider an S-Attributed grammar for constant expressions:
  - Each nonterminal has a single synthetic attribute: \( val \)
  - The annotated parse tree for \( 5 + 2 \times 3 \) is shown below

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E \rightarrow E^2 + T )</td>
<td>( E.val := E^2.val + T.val )</td>
</tr>
<tr>
<td>( E \rightarrow T )</td>
<td>( E.val := T.val )</td>
</tr>
<tr>
<td>( T \rightarrow T^2 \times F )</td>
<td>( T.val := T^2.val \times F.val )</td>
</tr>
<tr>
<td>( T \rightarrow F )</td>
<td>( T.val := F.val )</td>
</tr>
<tr>
<td>( F \rightarrow ( E ) )</td>
<td>( F.val := E.val )</td>
</tr>
<tr>
<td>( F \rightarrow \text{num} )</td>
<td>( F.val := \text{num}.val )</td>
</tr>
</tbody>
</table>

Semantic Rules

- \( E.val = 11 \)
- \( E.val = 5 \)
- \( T.val = 6 \)
- \( T.val = 5 \)
- \( T.val = 2 \)
- \( F.val = 3 \)
- \( F.val = 5 \)
- \( F.val = 2 \)
- \( \text{num}.val = 3 \)
- \( \text{num}.val = 5 \)
- \( \text{num}.val = 2 \)
Constructing Syntax Trees for Expressions

- A syntax tree is a condensed form of a parse tree
- A syntax tree can be used as an intermediate representation
- Each node is a structure with several fields
- To construct a syntax tree, we need …
  - `mknod(e(op, left, right))` creates a new node for a binary operator
    - `op` is a binary operator
    - `left` and `right` are pointers to the left and right subtrees
  - `idTable lookup(name)` searches the identifier table for a given `name`
    - Returns a pointer to the found identifier symbol
    - Returns NULL if `name` is not found
S-Attributed Grammar for Syntax Trees

- An S-attributed grammar is used for constructing a syntax tree
- A synthetic attribute `ptr` is used with `E`, `T`, `F` and `num`
  - `ptr` is a pointer that points at the syntax generated for `E`, `T`, and `F`
  - `ptr` is also used to point at a literal symbol for the token `num`

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>E \to E^2 + T</code></td>
<td><code>E.ptr := mknode('+', E^2.ptr, T.ptr)</code></td>
</tr>
<tr>
<td><code>E \to E^2 - T</code></td>
<td><code>E.ptr := mknode('-', E^2.ptr, T.ptr)</code></td>
</tr>
<tr>
<td><code>E \to T</code></td>
<td><code>E.ptr := T.ptr</code></td>
</tr>
<tr>
<td><code>T \to T^2 * F</code></td>
<td><code>T.ptr := mknode('*', T^2.ptr, F.ptr)</code></td>
</tr>
<tr>
<td><code>T \to T^2 / F</code></td>
<td><code>T.ptr := mknode('/', T^2.ptr, F.ptr)</code></td>
</tr>
<tr>
<td><code>T \to F</code></td>
<td><code>T.ptr := F.ptr</code></td>
</tr>
<tr>
<td><code>F \to ( E )</code></td>
<td><code>F.ptr := E.ptr</code></td>
</tr>
<tr>
<td><code>F \to id</code></td>
<td><code>F.ptr := idTable.lookup(id.name)</code></td>
</tr>
<tr>
<td><code>F \to num</code></td>
<td><code>F.ptr := num.ptr</code></td>
</tr>
</tbody>
</table>
L-Attributed Grammars

- Consider a typical production of the form: $A \rightarrow X_1 X_2 \ldots X_n$
- An attribute grammar is L-attributed if and only if:
  - Each inherited attribute of a right-hand-side symbol $X_j$ depends only on inherited attributes of $A$ and arbitrary attributes of the symbols $X_1, \ldots, X_{j-1}$
  - Each synthetic attribute of $A$ depends only on its inherited attributes and arbitrary attributes of the right-hand side symbols: $X_1 X_2 \ldots X_n$
- When Evaluating the attributes of an L-attributed production:
  - Evaluate the inherited attributes of $A$ (left-hand-side)
  - Evaluate the inherited then the synthesized attributes of $X_j$ from left to right
  - Evaluate the synthesized attribute of $A$
- If the underlying CFG is LL and L-attributed, we can evaluate the attributes in one pass by an LL Parser
- Every S-attributed grammar is also L-attributed
L-Attributed Grammar Evaluation

- L-attributed grammars are well-suited for LL-based evaluation
- Consider the prediction of production: $A \rightarrow X Y$
  - Evaluate and Push Inherited attributes of $A$: … Inh($A$)
  - Evaluate and Push Inherited attributes of $X$: … Inh($A$) Inh($X$)
  - Evaluate and Push Synthetic attributes of $X$ after parsing $X$:
    … Inh($A$) Inh($X$) Syn($X$)
  - Evaluate and Push Inherited attributes of $Y$:
    … Inh($A$) Inh($X$) Syn($X$) Inh($Y$)
  - Evaluate and Push Synthetic attributes of $Y$ after parsing $Y$:
    … Inh($A$) Inh($X$) Syn($X$) Inh($Y$) Syn($Y$)
  - Pop attributes of $X$ and $Y$ and push Synthetic attributes of $A$:
    … Inh($A$) Syn($A$)
- Attribute values are at known locations relative to stacktop
Example of an L-Attributed Grammar

- A C-like declaration generated by the non-terminal $D$ consists of
  - Keyword `int` or `float`, followed by a list of identifiers
- The non-terminal $T$ has a **synthesized attribute** `type`
- The non-terminal $L$ has an **inherited attribute** `type`
- The function `enter` creates a new symbol entry in a symbol table

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D \rightarrow T \ id \ L ; ;$</td>
<td>$enter(id.name, T.type)$</td>
</tr>
<tr>
<td>$L \rightarrow , \ id \ L^2$</td>
<td>$L.type := T.type$</td>
</tr>
<tr>
<td>$T \rightarrow int$</td>
<td>$T.type := INT_TYPE$</td>
</tr>
<tr>
<td>$T \rightarrow float$</td>
<td>$T.type := FLOAT_TYPE$</td>
</tr>
<tr>
<td>$L \rightarrow \epsilon$</td>
<td>$enter(id.name, L.type)$</td>
</tr>
<tr>
<td>$L^2.type := L.type$</td>
<td>$id_1 \rightarrow \text{float}$</td>
</tr>
</tbody>
</table>

Parse tree for: `float id , id , id ;`
Replacing Inherited Attributes by Synthesized Ones

- It is sometimes possible to avoid the use of inherited attributes
- This requires changing the underlying grammar
- Consider a Pascal-like declaration:
  - The first grammar uses an inherited attribute \( \text{type} \) for \( L \)
  - However, the first grammar is NOT \( L \)-attributed because \( L.\text{type} \) inherits the attribute of a right-sibling \( T \)
  - The second grammar is \( S \)-attributed. It uses synthetic attributes only.

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>( D \rightarrow L : T ; )</td>
<td>( L.\text{type} := T.\text{type} )</td>
<td>( D \rightarrow \text{id} L )</td>
<td>enter(\text{id}.name, ( L.\text{type} ))</td>
</tr>
<tr>
<td>( L \rightarrow L^2 , \text{id} )</td>
<td>enter(\text{id}.name, ( L.\text{type} )) ( L^2.\text{type} := L.\text{type} )</td>
<td>( L \rightarrow , \text{id} L^2 )</td>
<td>enter(\text{id}.name, ( L^2.\text{type} ))</td>
</tr>
<tr>
<td>( L \rightarrow \text{id} )</td>
<td>enter(\text{id}.name, ( L.\text{type} ))</td>
<td>( L \rightarrow : T ; )</td>
<td>( L.\text{type} := T.\text{type} )</td>
</tr>
<tr>
<td>( T \rightarrow \text{integer} )</td>
<td>( T.\text{type} := \text{integer} )</td>
<td>( T \rightarrow \text{integer} )</td>
<td>( T.\text{type} := \text{integer} )</td>
</tr>
<tr>
<td>( T \rightarrow \text{real} )</td>
<td>( T.\text{type} := \text{real} )</td>
<td>( T \rightarrow \text{real} )</td>
<td>( T.\text{type} := \text{real} )</td>
</tr>
</tbody>
</table>