Chapter 4 - Semantic Analysis
Role of Semantic Analysis

- Following parsing, the next two phases of the "typical" compiler are:
  - Semantic analysis.
  - (intermediate) Code generation.
- The principal job of the semantic analyzer is to enforce static semantic rules.
  - Constructs a syntax tree (usually first).
  - Information gathered is needed by the code generator.
Role of Semantic Analysis

- There is considerable variety in the extent to which parsing, semantic analysis, and intermediate code generation are interleaved.
- A common approach interleaves construction of a syntax tree with parsing (no explicit parse tree), and then follows with separate, sequential phases for semantic analysis and code generation.
Role of Semantic Analysis

- The PL/0 compiler has no optimization to speak of (there's a tiny little trivial phase, which operates on the syntax tree).
- Its code generator produces MIPs assembler, rather than a machine-independent intermediate form.
Static Semantics

- *Syntax* concerns the form of a valid program, while *semantics* concerns its meaning
  - Context-free grammars are not powerful enough to describe certain rules, e.g. checking variable declaration with variable use.
- *Static semantic* rules are enforced by a compiler at compile time.
  - Implemented in semantic analysis phase of the compiler.
- Examples:
  - Type checking.
  - Identifiers are used in appropriate context.
  - Check subroutine call arguments.
  - Check labels.
Dynamic Semantics

- *Dynamic semantic rules* are enforced by the compiler by generating code to perform the checks at run-time
- Examples:
  - Array subscript values are within bounds
  - Arithmetic errors
  - Pointers are not dereferenced unless pointing to valid object
  - A variable is used but hasn't been initialized
- Some languages (Euclid, Eiffel) allow programmers to add explicit dynamic semantic checks in the form of assertions, e.g. `assert denominator not= 0`
- When a check fails at run time, an exception is raised
Attribute Grammars

- Both semantic analysis and (intermediate) code generation can be described in terms of annotation, or "decoration" of a parse or syntax tree.
- **Attribute grammars** provide a formal framework for **decorating** such a tree.
- The notes below discuss attribute grammars and their ad-hoc cousins, **action routines**.
Attribute Grammars

- An attribute grammar “connects” syntax with semantics
- Each grammar production has a semantic rule with actions (e.g. assignments) to modify values of attributes of (non)terminals
  - A (non)terminal may have any number of attributes
  - Attributes have values that hold information related to the (non)terminal
- General form:

  production   semantic rule
  \(<A> ::= <B> <C>\)    \(A.a := \ldots; B.a := \ldots; C.a := \ldots\)

- Semantic rules are used by a compiler to enforce static semantics and/or to produce an abstract syntax tree while parsing tokens
  - Syntax directed translation.
- Can also be used to build simple language interpreters
Attribute Grammars

• We'll start with decoration of parse trees, then consider syntax trees.
• Consider the following LR (bottom-up) grammar for arithmetic expressions made of constants, with precedence and associativity:

\[
\begin{align*}
E & \rightarrow E + T \\
E & \rightarrow E - T \\
E & \rightarrow T \\
T & \rightarrow T * F \\
T & \rightarrow T / F \\
T & \rightarrow F \\
F & \rightarrow - F \\
F & \rightarrow (E) \\
F & \rightarrow \text{const}
\end{align*}
\]

This says nothing about what the program means.
Attribute Grammars

- We can turn this into an attribute grammar as follows:

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic</th>
</tr>
</thead>
<tbody>
<tr>
<td>E → E + T</td>
<td>E1.val = E2.val + T.val</td>
</tr>
<tr>
<td>E → E - T</td>
<td>E1.val = E2.val - T.val</td>
</tr>
<tr>
<td>E → T</td>
<td>E.val = T.val</td>
</tr>
<tr>
<td>T → T * F</td>
<td>T1.val = T2.val * F.val</td>
</tr>
<tr>
<td>T → T / F</td>
<td>T1.val = T2.val / F.val</td>
</tr>
<tr>
<td>T → F</td>
<td>T.val = F.val</td>
</tr>
<tr>
<td>F → - F</td>
<td>F1.val = - F2.val</td>
</tr>
<tr>
<td>F → (E)</td>
<td>F.val = E.val</td>
</tr>
<tr>
<td>F → const</td>
<td>F.val = C.val</td>
</tr>
</tbody>
</table>
Attribute Grammars

- The attribute grammar serves to define the semantics of the input program.
- Attribute rules are best thought of as definitions, not assignments.
- They are not necessarily meant to be evaluated at any particular time, or in any particular order, though they do define their left-hand side in terms of the right-hand side.
Evaluating Attributes

• The process of evaluating attributes is called annotation, or decoration, of the parse tree.
  ▫ When a parse tree under this grammar is fully decorated, the value of the expression will be in the val attribute of the root.
  ▫ The code fragments for the rules are called semantic functions.

• Strictly speaking, they should be cast as functions, e.g.,
  \( E1.val = \text{sum} \ (E2.val, \ T.val) \)

• The val attribute of a (non)terminal holds the subtotal value of the subexpression.

• Nonterminals are indexed in the attribute grammar to distinguish multiple occurrences of the nonterminal in a production.
Evaluating Attributes (indexed)

production

\[ <E_1> ::= <E_2> + <T> \]
\[ <E_1> ::= <E_2> - <T> \]
\[ <E> ::= <T> \]
\[ <T_1> ::= <T_2> * <F> \]
\[ <T_1> ::= <T_2> / <F> \]
\[ <T> ::= <F> \]
\[ <F_1> ::= - <F_2> \]
\[ <F> ::= ( <E> ) \]
\[ <F> ::= \text{unsigned\_int} \]

semantic rule

\[ E_1.\text{val} := E_2.\text{val} + T.\text{val} \]
\[ E_1.\text{val} := E_2.\text{val} - T.\text{val} \]
\[ E.\text{val} := T.\text{val} \]
\[ T_1.\text{val} := T_2.\text{val} * F.\text{val} \]
\[ T_1.\text{val} := T_2.\text{val} / F.\text{val} \]
\[ T.\text{val} := F.\text{val} \]
\[ F_1.\text{val} := -F_2.\text{val} \]
\[ F.\text{val} := E.\text{val} \]
\[ F.\text{val} := \text{unsigned\_int.\text{val}} \]
Decorated Parse Trees

- A parser produces a parse tree that is *decorated* with the attribute values.
- Example decorated parse tree of \((1+3)\times2\) with the `val` attributes.
Evaluating Attributes

- This is a very simple attribute grammar:
  - Each symbol has at most one attribute.
    - The punctuation marks have no attributes.
- These attributes are all so-called *synthetized* attributes:
  - They are calculated only from the attributes of things below them in the parse tree.
Synthesized Attributes

- *Synthesized attributes* of a node hold values that are computed from attribute values of the *child* nodes in the parse tree and therefore information flows *upwards*.

**production**

\[ <E_1> ::= <E_2> + <T> \]

**semantic rule**

\[ E_1.val := E_2.val + T.val \]
Evaluating Attributes

• In general, we are allowed both synthesized and inherited attributes:
  ▫ Inherited attributes may depend on things above or to the side of them in the parse tree.
  ▫ Tokens have only synthesized attributes, initialized by the scanner (name of an identifier, value of a constant, etc.).
  ▫ Inherited attributes of the start symbol constitute run-time parameters of the compiler.
Inherited Attributes

- *Inherited attributes* of *child* nodes are set by the *parent* node or sibling nodes and therefore information flows *downwards*.

**Production**

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>D -&gt; T L</td>
<td>L.in = T.type</td>
</tr>
<tr>
<td>T-&gt;int</td>
<td>T.type = integer</td>
</tr>
<tr>
<td>T-&gt;real</td>
<td>T.type = real</td>
</tr>
<tr>
<td>L-&gt;L1, id</td>
<td>L1.in = L.in, addtype(id.entry, L.in)</td>
</tr>
<tr>
<td>L-&gt;id</td>
<td>addtype(id.entry, L.in)</td>
</tr>
</tbody>
</table>

real id1, id2, id3
Inherited Attributes

• Another example:

Production

\[<E> ::= <T> <TT>\]

\[<TT_1> ::= + <T> <TT_2>\]

\[<TT> ::= \varepsilon\]

Semantic rule

\[TT.st := T.val;\]
\[E.val := TT.val\]
\[TT_2.st := TT_1.st + T.val;\]
\[TT_1.val := TT_2.val\]
\[TT.val := TT.st\]
Evaluating Attributes

- The grammar above is called S-attributed because it uses only synthesized attributes.
- Attribute flow (attribute dependence graph) is purely bottom-up.
- It is SLR(1), but not LL(1).
- An equivalent LL(1) grammar requires inherited attributes:
Attribute Flow

• An *attribute flow algorithm* propagates attribute values through the parse tree by traversing the tree according to the *set* (write) and *use* (read) dependencies (an attribute must be set before it is used)

production

\[ <E> ::= <T> <TT> \]

semantic rule

\[ TT.st := T.val \]
Attribute Flow

Production

\[ <TT_1> ::= + <T> <TT_2> \]

Semantic rule

\[ TT_2.st := TT_1.st + T.val \]
Attribute Flow

Production

\[ <TT> ::= \varepsilon \]

Semantic rule

\[ TT.val := TT.st \]
Attribute Flow

Production

\(<TT_1> ::= + <T> <TT_2>\)

Semantic rule

\(TT_1.val := TT_2.val\)
Attribute Flow

Production
\[ <E> ::= <T> <TT> \]

Semantic rule
\[ E.val := TT.val \]
S- and L-Attributed Grammars

• A grammar is called *S-attributed* if all attributes are synthesized

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>L → E n</td>
<td>print(E.val)</td>
</tr>
<tr>
<td>E → E1 + T</td>
<td>E.val = E1.val + T.val</td>
</tr>
<tr>
<td>E → T</td>
<td>E.val = T.val</td>
</tr>
<tr>
<td>T → T1 * F</td>
<td>T.val = T1.val * F.val</td>
</tr>
<tr>
<td>T → F</td>
<td>T.val = F.val</td>
</tr>
<tr>
<td>F → (E)</td>
<td>F.val = E.val</td>
</tr>
<tr>
<td>F → digits</td>
<td>F.val = digits.lexval</td>
</tr>
</tbody>
</table>
S- and L-Attributed Grammars

- A grammar is called *L-attributed* if the parse tree traversal to update attribute values is always left-to-right and depth-first.
  - For a production $A \rightarrow X_1 \ X_2 \ X_3 \ldots \ X_n$
    - The attributes of $X_j$ ($1 \leq j \leq n$) only depends on
      - The attributes of $X_1, X_2, \ldots, X_{j-1}$
      - The inherited attributed of $A$.
  - Values of inherited attributes must be passed down to children from left to right.
  - Semantic rules can be applied immediately during parsing and parse trees do not need to be kept in memory.
  - This is an essential grammar property for a one-pass compiler.
- An S-attributed grammar is a special case of an L-attributed grammar.
Evaluating Attributes– Example

• Attribute grammars:
  ▫ This attribute grammar is a good bit messier than the first one, but it is still L-attributed, which means that the attributes can be evaluated in a single left-to-right pass over the input.
  ▫ In fact, they can be evaluated during an LL parse.
  ▫ Each synthetic attribute of a LHS symbol (by definition of synthetic) depends only on attributes of its RHS symbols.
  ▫ Each inherited attribute of a RHS symbol (by definition of L-attributed) depends only on:
    • Inherited attributes of the LHS symbol.
    • Synthetic or inherited attributes of symbols to its left in the RHS.

• L-attributed grammars are the most general class of attribute grammars that can be evaluated during an LL parse.
Evaluating Attributes

• There are certain tasks, such as generation of code for short-circuit Boolean expression evaluation, that are easiest to express with non-L-attributed attribute grammars.

• Because of the potential cost of complex traversal schemes, however, most real-world compilers insist that the grammar be L-attributed.
Action Routines

• We can tie this discussion back into the earlier issue of separated phases v.s. on-the-fly semantic analysis and/or code generation.
• If semantic analysis and/or code generation are interleaved with parsing, then the translation scheme we use to evaluate attributes must be L-attributed.
• If we break semantic analysis and code generation out into separate phase(s), then the code that builds the parse/syntax tree must still use a left-to-right (L-attributed) translation scheme.
• The later phases are free to use a fancier translation scheme if they want.
Action Routines

- There are automatic tools that generate translation schemes for context-free grammars or tree grammars (which describe the possible structure of a syntax tree).
  - These tools are heavily used in syntax-based editors and incremental compilers.
  - Most ordinary compilers, however, use ad-hoc techniques.
Action Routines

- An ad-hoc translation scheme that is interleaved with parsing takes the form of a set of action routines:
  - An action routine is a semantic function that we tell the compiler to execute at a particular point in the parse.
- If semantic analysis and code generation are interleaved with parsing, then action routines can be used to perform semantic checks and generate code.
- If semantic analysis and code generation are broken out as separate phases, then action routines can be used to build a syntax tree.
  - A parse tree could be built completely automatically.
  - We wouldn't need action routines for that purpose.
Action Routines

- Later compilation phases can then consist of ad-hoc tree traversal(s), or can use an automatic tool to generate a translation scheme.
  - The PL/0 compiler uses ad-hoc traversals that are almost (but not quite) left-to-right.
- For our LL(1) attribute grammar, we could put in explicit action routines as follows:
Example L-Attributed Grammar

- Implementing a calculator

Production

\[
\begin{align*}
<E> & ::= <T> <TT> \\
<TT_1> & ::= + <T> <TT_2> \\
<TT_1> & ::= - <T> <TT_2> \\
<TT> & ::= \varepsilon \\
<T> & ::= <F> <FT> \\
<FT_1> & ::= * <F> <FT_2> \\
<FT_1> & ::= / <F> <FT_2> \\
<FT> & ::= \varepsilon \\
<F_1> & ::= - <F_2> \\
<F> & ::= ( <E> ) \\
<F> & ::= unsigned_int
\end{align*}
\]

Semantic rule

\[
\begin{align*}
TT.st & ::= T.val; \\
E.val & ::= TT.val \\
TT_2.st & ::= TT_1.st + T.val; \\
TT_1.val & ::= TT_2.val \\
TT_2.st & ::= TT_1.st - T.val; \\
TT_1.val & ::= TT_2.val \\
TT.val & ::= TT.st \\
FT.st & ::= F.val; \\
T.val & ::= FT.val \\
FT_2.st & ::= FT_1.st * F.val; \\
FT_1.val & ::= FT_2.val \\
FT_2.st & ::= FT_1.st / F.val; \\
FT_1.val & ::= FT_2.val \\
FT.val & ::= FT.st \\
F_1.val & ::= -F_2.val \\
F.val & ::= E.val \\
F.val & ::= unsigned_int.val
\end{align*}
\]
Constructing Abstract Syntax Trees with Attribute Grammars

- Three operations to create nodes for an AST tree that represents expressions:
  - `mk_bin_op(op, left, right)`: constructs a new node that contains a binary operator `op` and AST sub-trees `left` and `right` representing the operator’s operands and returns pointer to the new node.
  - `mk_un_op(op, node)`: constructs a new node that contains a unary operator `op` and sub-tree `node` representing the operator’s operand and returns pointer to the new node.
  - `mk_leaf(value)`: constructs an AST leaf that contains a value and returns pointer to the new node.
Constructing AST with Attribute Grammar

E->E1+ E2
E->E1 - E2
E->E1* E2
E->E1 / E2
E->(E1)
E-> -E1
E->number

E.ptr = mk_bin_op('+', E1.ptr, E2.ptr);
E.ptr = mk_bin_op('-', E1.ptr, E2.ptr);
E.ptr = mk_bin_op('*', E1.ptr, E2.ptr);
E.ptr = mk_bin_op('/', E1.ptr, E2.ptr);
E.ptr = E1.ptr
E.ptr = mk_un_op('-', E1.ptr);
E.ptr = mk_leaf(number.val);
Action Routines

\[
\begin{align*}
E & \rightarrow \ T \ \{ \ TT.st := T.ptr \ \} \ TT \ \{ \ E.ptr := TT.ptr \ \} \\
TT_1 & \rightarrow \ + \ T \ \{ \ TT_2.st := \text{make}_\text{bin}_\text{op}("+", \ TT_1.st, T.ptr) \ \} \ TT_2 \ \{ \ TT_1.ptr := TT_2.ptr \ \} \\
TT_1 & \rightarrow \ - \ T \ \{ \ TT_2.st := \text{make}_\text{bin}_\text{op}("-", \ TT_1.st, T.ptr) \ \} \ TT_2 \ \{ \ TT_1.ptr := TT_2.ptr \ \} \\
TT & \rightarrow \ \epsilon \ \{ \ TT.ptr := TT.st \ \} \\
T & \rightarrow \ F \ \{ \ FT.st := F.ptr \ \} \ FT \ \{ \ T.ptr := FT.ptr \ \} \\
FT_1 & \rightarrow \ * \ F \ \{ \ FT_2.st := \text{make}_\text{bin}_\text{op}("\times", \ FT_1.st, F.ptr) \ \} \ FT_2 \ \{ \ FT_1.ptr := FT_2.ptr \ \} \\
FT_1 & \rightarrow \ / \ F \ \{ \ FT_2.st := \text{make}_\text{bin}_\text{op}("\div", \ FT_1.st, F.ptr) \ \} \ FT_2 \ \{ \ FT_1.ptr := FT_2.ptr \ \} \\
FT & \rightarrow \ \epsilon \ \{ \ FT.ptr := FT.st \ \} \\
F_1 & \rightarrow \ - \ F_2 \ \{ \ F_1.ptr := \text{make}_\text{un}_\text{op}("+/\_\", \ F_2.ptr) \ \} \\
F & \rightarrow \ ( \ E \ ) \ \{ \ F.ptr := E.ptr \ \} \\
F & \rightarrow \ \text{const} \ \{ \ F.ptr := \text{make}_\text{leaf}(\text{const}.ptr) \ \}
\end{align*}
\]

Figure 4.9 LL(1) grammar with action routines to build a syntax tree.
Space Management for Attributes

- Entries in the attributes stack are pushed and popped automatically.

```
program  →  stmt_list $$
stmt_list →  stmt_list decl | stmt_list stmt | ε
decl  →  int id | real id
stmt  →  id := expr | read id | write expr
expr  →  term | expr add_op term
term  →  factor | term mult_op factor
factor →  ( expr ) | id | int_const | real_const | 
       | float ( expr ) | trunc ( expr )
add_op  →  + | -
mult_op  →  * | /
```

**Figure 4.11** Context-free grammar for a calculator language with types and declarations. The intent is that every identifier be declared before use, and that types not be mixed in computations.
Decorating a Syntax Tree

- Syntax tree for a simple program to print an average of an integer and a real.

```
program
  int_decl
    read
      a
  real_decl
    read
      a
      b
    write
      b
      null
  int a
  read a
  real b
  read b
write (float (a) + b) / 2.0
```

Figure 4.12  Syntax tree for a simple calculator program.
Decorating a Syntax Tree

\[
\begin{align*}
\text{program} & \rightarrow \text{item} \\
\text{int\_decl} : \text{item} & \rightarrow \text{id} \text{ item} \\
\text{read} : \text{item} & \rightarrow \text{id} \text{ item} \\
\text{real\_decl} : \text{item} & \rightarrow \text{id} \text{ item} \\
\text{write} : \text{item} & \rightarrow \text{expr} \text{ item} \\
\text{null} : \text{item} & \rightarrow \epsilon \\
\div' : \text{expr} & \rightarrow \text{expr} \text{ expr} \\
+ : \text{expr} & \rightarrow \text{expr} \text{ expr} \\
\text{float} : \text{expr} & \rightarrow \text{expr} \\
\text{id} : \text{expr} & \rightarrow \epsilon \\
\text{real\_const} : \text{expr} & \rightarrow \epsilon
\end{align*}
\]

- Tree grammar representing structure of syntax tree in Figure 4.12.
Decorating a Syntax Tree

- Sample of complete tree grammar representing structure of syntax tree in Figure 4.12.

```plaintext
id : expr —> e
    ▷ if (id.name, A) ∈ expr.symtab         -- for some type A
        expr.errors := null
        expr.type := A
    else
        expr.errors := [id.name "undefined at" id.location]
        expr.type := error

int_const : expr —> e
    ▷ expr.type := int

real_const : expr —> e
    ▷ expr.type := real

`+` : expr1 —> expr2 expr3
    ▷ expr2.symtab := expr1.symtab
    ▷ expr3.symtab := expr1.symtab
    ▷ check_types(expr1, expr2, expr3)

`-` : expr1 —> expr2 expr3
    ▷ expr2.symtab := expr1.symtab
    ▷ expr3.symtab := expr1.symtab
    ▷ check_types(expr1, expr2, expr3)

`*` : expr1 —> expr2 expr3
    ▷ expr2.symtab := expr1.symtab
    ▷ expr3.symtab := expr1.symtab
    ▷ check_types(expr1, expr2, expr3)

`/` : expr1 —> expr2 expr3
    ▷ expr2.symtab := expr1.symtab
    ▷ expr3.symtab := expr1.symtab
    ▷ check_types(expr1, expr2, expr3)

float : expr1 —> expr2
    ▷ expr2.symtab := expr1.symtab
    ▷ convert_type(expr2, expr1, int, real, "float of non-int")

trunc : expr1 —> expr2
    ▷ expr2.symtab := expr1.symtab
    ▷ convert_type(expr2, expr1, real, int, "trunc of non-real")
```