Semantic Processing: Syntax Directed Translation

- **Attributes**: Associate information with language constructs by attaching attributes to grammar symbols representing that construct.
  
  An attribute can represent anything (reasonable) that we choose, e.g. a string, number, type, memory location, code fragment etc.

- **Semantic rules**: Values for attributes are computed using **semantic rules** associated with grammar productions.
  
  A parse tree showing the values of attributes at each node is called an **annotated parse tree**.

**Example**: Attributes for an Identifier

name : character string, obtained from scanner.

scope

type :
  - integer
  - array :
    - no. of dimensions
    - upper and lower bounds for each dimension
    - type of elements
  - record :
    - name and type of each field
  - function
    - no. of parameters
    - types of parameters (in order)
    - type of returned value
    - entry point in memory
    - size of stack frame
Example: Associating Semantic Rules with Productions

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E \rightarrow E_1 + T$</td>
<td>$E.val := E_1.val \oplus T.val$</td>
</tr>
<tr>
<td>$E \rightarrow T$</td>
<td>$E.val := T.val$</td>
</tr>
<tr>
<td>$T \rightarrow T_1 * F$</td>
<td>$T.val := T_1.val \otimes F.val$</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>$F \rightarrow \text{intcon}$</td>
<td>$F.val := \text{intcon}.val$</td>
</tr>
</tbody>
</table>

Note: The semantic rules also impose an evaluation order on the attributes.

Two-Pass vs. One-Pass Compilation

Two-Pass

1. Parse input and use semantic rules to:
   (a) process declarations into symbol table
   (b) construct syntax tree

2. Traverse syntax tree:
   (a) check types
   (b) make storage allocation decisions
   (c) generate code
One-Pass

1. Parse input and use semantic rules to:
   (a) process declarations into symbol table
   (b) check types
   (c) make storage allocation decisions
   (d) generate code

Inherited and Synthesized Attributes

Inherited Attributes: An attribute at a node is inherited if its value is computed from attribute values at the siblings and/or parent of that node in the parse tree.
Synthesized Attributes: An attribute at a node is synthesized if its value is computed from the attribute values of the children of that node in the parse tree.

6.1.1. Attribute Grammars

Basic Idea:
- Every grammar symbol is associated with a set of attributes.
- Semantic rules specify how each attribute is to be computed.

The attributes of a grammar symbol are partitioned into two sets: inherited and synthesized. I.e., for any particular grammar symbol, a given attribute cannot be inherited in some places and synthesized in others.
E.g.: $A \rightarrow X \ Y \ Z$

**S-Attributed Grammars**

*Definition*: Grammar containing only synthesized attributes is called *S-attributed*.

- Synthesized attributes can be conveniently handled during bottom up parsing as it builds the parse tree bottom up.

**L-Attributed Grammars**

*Definition*: Grammar for which the attributes can always be evaluated by a depth-first L-to-R traversal of the parse tree.

- All attributes can be conveniently handled during LL(1) parsing because the parse tree is built depth-first L-to-R.
- Every S-attributed definition is L-attributed.
Example

We will develop semantic rules for constructing symbol table from the declarations and constructing syntax tree for the expression.

- *Inherited attribute* needed to propagate the type to each declared variable.

- *Synthesized attribute* needed to construct syntax tree for an expression from syntax trees of subexpressions.

```
int a, b, c;
a + b * c;
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Addr</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>int</td>
<td>1</td>
<td>[1]</td>
</tr>
<tr>
<td>b</td>
<td>int</td>
<td>2</td>
<td>[2]</td>
</tr>
<tr>
<td>c</td>
<td>int</td>
<td>3</td>
<td>[3]</td>
</tr>
</tbody>
</table>

Syntax Tree:

```
+    *
[1]  [2]  [3]
```
Syntax Trees

Programs

Declarations <-> Computations

Symbol Table <-> Syntax Tree

- A syntax tree is a tree that shows the syntactic structure of a program, while omitting irrelevant detail present in a parse tree.

- Each node of a syntax tree represents "what to do" at that point, i.e., a computation. The children of the node correspond to the objects to which that computation is applied.

Example

Grammar:

\[
E \rightarrow E + T \mid T \\
T \rightarrow T \ast F \mid F \\
F \rightarrow (E) \mid id
\]

Input: id + id * id

Parse Tree:

\[
\begin{align*}
& \quad E \\
& \quad \quad \quad E \\
& \quad \quad \quad \quad T \\
& \quad \quad \quad \quad \quad T \\
& \quad \quad \quad \quad \quad F \\
& \quad \quad \quad \quad \quad \quad F \\
& \quad \quad \quad \quad \quad \quad \quad id + id \ast id
\end{align*}
\]

Syntax Tree:

\[
\begin{align*}
& \quad + \\
& \quad \quad * \\
& \quad \quad \quad id \\
& \quad \quad \quad \quad id \\
& \quad \quad \quad \quad id \\
& \quad \quad id + id \ast id
\end{align*}
\]
Structure of Syntax Trees

Expression:
- Leaves: identifiers or constants.
- Internal nodes labelled with operations.
- Children of a node are its operands.

Statements:
- A node’s label indicates what kind of statement it is.
- The children of a node correspond to the components of the statement.

While

if

:=

;

Cond  Body  Cond  Then  Else  LHS  RHS  Stmt  StmtRest

6.3. Symbol Tables

Purpose: To hold information about identifiers that is computed at one point and looked up at later points during compilation.

Example: type of a variable; entry point for a function.

Operations: insert, lookup, delete.

Common implementations: linked lists, hash tables.
Managing Scope Information

- When a name is looked up in a symbol table, the entry for the “appropriate” declaration of that name must be returned.
  The scope rules of the language determine which declaration is appropriate.

- Often, the appropriate declaration for a name is the “most closely nested” one. A simple implementation of this is to push a new symbol table when entering a new scope, and pop it when leaving it:
  - Implement the stack of symbol tables as a linked list of tables.
  - lookup: search backward starting at the innermost scope.
  - insert, delete: works on the innermost scope.

- Information may be “deleted” when leaving a scope; but it may be necessary to retain this information for use by run-time tools, e.g. debuggers.

Processing Declarations

Goal: Store information about variable names and types in symbol table.

Use of Attributes: To propagate type information to the various identifiers appearing in a declaration.

\[
\begin{align*}
Decl & \rightarrow Type \ Id_list ; \\
Id_list & \rightarrow id , Id_list \mid id \\
Type & \rightarrow int \mid real
\end{align*}
\]
**Semantic Rules:**

- *Type* synthesizes the value of *tval*;

- *Id_list* uses *tval* as an inherited attribute; defines type information in symbol table entries corresponding to *id*.

**Production**

**Semantic Rule**

\[
\begin{align*}
\text{Decl} & \rightarrow \text{Type } \text{Id_list} ; \\
\text{Id_list} & \rightarrow \text{id } , \text{Id_list} \\
\text{id.type} & := \text{Id_list.tval;} \\
\text{symtab.insert(id.name.id.type)} \\
\text{Id_list.tval} & := \text{Id_list.tval} \\
\text{Type} & \rightarrow \text{int} \\
\text{Type.tval} & = \text{int} \\
\text{Type} & \rightarrow \text{real} \\
\text{Type.tval} & = \text{real}
\end{align*}
\]

**Semantic Rules for Constructing Expression Syntax Tree**

**Goal**: Construct syntax tree for the expression; associate references to *ids* by entries in symbol table.

**Use of Attributes**: To propagate syntax trees for smaller subexpressions needed to from syntax trees for larger expressions.

**Production**

**Semantic Rule**

\[
\begin{align*}
E & \rightarrow E_1 + T \\
E.tree & = \text{mktree(PLUS, } E_1.tree, T.tree) \\
E & \rightarrow T \\
E.tree & = T.tree
\end{align*}
\]
\[ T \rightarrow T \star F \]
\[ T.tree = \text{mktree} (\text{TIMES}, T.tree, F.tree) \]

\[ T \rightarrow F \]
\[ T.tree = F.tree \]

\[ F \rightarrow \text{id} \]
\[ F.tree = \text{mknode} (\text{idnode}, \text{symtab\_lookup}(\text{id}\_name)) \]

\[ F \rightarrow \text{intconst} \]
\[ F.tree = \text{mknode} (\text{intconstnode}, \text{intconst\_value}) \]

---

**Syntax-Directed Definitions vs. Translation Schemes**

*Syntax-directed definitions* describe relationships among attributes associated with grammar symbols (so far we have only looked at these).

*Syntax-directed translation schemes* describe the order and timing of attribute computation.

- Embeds semantic rules into the grammar.
- Each semantic rule can only use information computed by already executed semantic rules.
Translation Scheme with Synthesized Attributes

- Synthesized attributes of a terminal are contained in the terminal symbol itself.

- Synthesized attribute associated with a non-terminal symbol is computed after seeing everything it derives.

\[
E \rightarrow E_1 \ + \ T \ \{E.tree = mktree(\text{PLUS}, E_1.tree, T.tree)\} \\
E \rightarrow T \ \{E.tree = T.tree\} \\
T \rightarrow T \ * \ F \ \{T.tree = mktree(\text{TIMES}, T.tree, F.tree)\} \\
T \rightarrow F \ \{T.tree = F.tree\} \\
F \rightarrow \text{id} \ \{F.tree = mknod\text{e}(\text{idnode}, \text{symtab.lookup}(\text{id.name}))\} \\
F \rightarrow \text{intconst} \ \{F.tree = mknod\text{e}(\text{intconstnode}, \text{intconst.value})\}
\]

Translation Scheme with Inherited Attributes

- Inherited attribute associated with a non-terminal is computed before encountering the non-terminal.

\[
E \rightarrow T \ \{R.tree = T.tree\} \\
\quad \{R.tree = R.tree\} \\
R \rightarrow + T \ \{R.tree = mktree(\text{PLUS}, R.tree, T.tree)\} \\
\quad \{R.tree = R.tree\} \\
R \rightarrow T \ \{R.tree = mktree(\text{TIMES}, R.tree, T.tree)\} \\
\quad \{R.tree = R.tree\} \\
R \rightarrow R \ \{R.tree = R.tree\} \\
T \rightarrow \text{id} \ \{T.tree = mknod\text{e}(\text{idnode}, \text{symtab.lookup}(\text{id.name}))\} \\
T \rightarrow \text{intconst} \ \{T.tree = mknod\text{e}(\text{intconstnode}, \text{intconst.value})\}
\]
Implementation Issues

*Triggering execution* of semantic actions: How can parsing actions be made to trigger execution of semantic rules?

*Managing and accessing attribute values*: Where should the attribute values be held and how should they be accessed?

Note: Solutions vary according to the type of parses: bottom-up vs. top-down.
Triggering Semantic Actions in a Bottom-Up Parser

- A reduction occurs in the parser at each point where a synthesized attribute is to be computed because computation of a synthesized attribute is performed at the end of the right hand side of a production.

\[ E \rightarrow E_1 + T \{ E.tree = mktree(\"+\", E_1.tree, T.tree) \} \]

Example

Reductions trigger execution of code corresponding to semantic rules.

- The same is not true for inherited attributes as semantic rules for their evaluation is embedded inside the right hand side of a production.

Augment the grammar with marker non-terminals to introduce reductions corresponding to evaluations of inherited attributes.

Example

Before transformation:

\[
\begin{align*}
E & \rightarrow T E' \\
E' & \rightarrow + T \{ \text{print } \texttt{+}\} E' \mid - T \{ \text{print } \texttt{-}\} E' \mid T \\
T & \rightarrow \text{num} \{ \text{print \texttt{num.val}} \}
\end{align*}
\]

After transformation:

\[
\begin{align*}
E & \rightarrow T E' \\
E' & \rightarrow + T M_1 E' \mid - T M_2 E' \mid T \\
T & \rightarrow \text{num} \{ \text{print \texttt{num.val}} \} \\
M_1 & \rightarrow \varepsilon \{ \text{print } \texttt{+}\} \\
M_2 & \rightarrow \varepsilon \{ \text{print } \texttt{-}\}
\end{align*}
\]
Managing Attributes in a Bottom-Up Parser

- A bottom-up parser maintains a semantic stack that parallels the syntax stack. Given a symbol $X$ in the syntax stack, the attributes of $X$ are stored in the corresponding position of the semantic stack.

- When a reduction is made, compute new synthesized attributes from the values currently on top of the stack.

- Computation of inherited attributes requires "reaching into" the semantic stack. We must ensure that the position that we must reach into is predictable.

Example with Synthesized Attribute:

\[
E \rightarrow E_1 + T \{
    y := \text{semantic\_stack}[top];
    x := \text{semantic\_stack}[top - 2];
    z := \text{mktree}(\text{\char121}, x, y);
    \text{semantic\_stack}[top - 2] := z;
    top := top - 2;
\}
\]

Example with Inherited Attribute:

\[
\begin{align*}
E &\rightarrow T E' \\
E' &\rightarrow OP T M E' \mid T \\
OP &\rightarrow + \mid - \\
M &\rightarrow \epsilon \quad \{\text{print semantic\_stack[top-2]}\} \\
T &\rightarrow \text{num} \quad \{\text{print num\_val}\}
\end{align*}
\]
During computation of an inherited attribute, the position of the desired attribute value in the semantic stack may not always be predictable, e.g.:

\[
\begin{align*}
S & \rightarrow a \ A \ C \quad \{C.i := A.s\} \\
S & \rightarrow b \ A \ B \ C \quad \{C.i := A.s\} \\
C & \rightarrow c \quad \{C.s := g(C.i)\}
\end{align*}
\]

In this case, there may or may not be a \( B \) between \( A \) and \( C \).

**Solution**: use additional marker that make the position predictable.

---

**Example** (cont’d)

**Original Grammar**:

\[
\begin{align*}
S & \rightarrow a \ A \ C \mid b \ A \ B \ C \\
C & \rightarrow c
\end{align*}
\]

**Transformed Grammar I**:

\[
\begin{align*}
S & \rightarrow a \ A \ C \quad \{C.i := A.s\} \\
S & \rightarrow b \ A \ B \ M \ C \quad \{M.i := A.s; C.i := M.s\} \\
C & \rightarrow c \quad \{C.s := g(C.i)\} \\
M & \rightarrow \varepsilon \quad \{M.s := M.i\}
\end{align*}
\]

**Transformed Grammar II**:

\[
\begin{align*}
S & \rightarrow a \ A \ M \ C \quad \{C.i := A.s\} \\
S & \rightarrow b \ A \ B \ C \quad \{C.i := A.s\} \\
C & \rightarrow c \quad \{C.s := g(C.i)\} \\
M & \rightarrow \varepsilon \quad \{
\}
\end{align*}
\]
Triggering Semantic Actions in a LL(1) Parser

- Unlike the bottom-up parser, there are no distinct parsing events which can be used to trigger the execution of semantic actions.

- Augment the grammar with *marker non-terminals* whose only purpose is to trigger execution of semantic actions.

  When a production rule is applied, these markers are pushed along with the rest of the symbols on to the syntax stack in reverse order.

  When a marker is popped from the syntax stack, the corresponding semantic action is executed.

Managing Attributes in a LL(1) Parser

- The syntax stack does not parallel the semantic stack – syntax stack contains what we expect to see in the future while the semantic stack contains attributes of constructs that have already been seen.

- For each production applied, reserve positions in the semantic stack to hold attributes for the left hand side non-terminal and right hand side symbols.

  Save these positions in the syntax stack to allow access to attributes.

  For more details see separate handout given in the class.