Syntax-Directed Translation

ASU Textbook Chapter 5

Tsan-sheng Hsu

tshsu@iis.sinica.edu.tw

http://www.iis.sinica.edu.tw/~tshsu
What is syntax-directed translation?

- **Definition:**
  - The compilation process is driven by the syntax.
  - The semantic routines perform interpretation based on the syntax structure.
  - Attaching **attributes** to the grammar symbols.
  - Values for attributes are computed by **semantic rules** associated with the grammar productions.
Example: Syntax-directed translation

- Example in a parse tree:
  - Annotate the parse tree by attaching semantic attributes to the nodes of the parse tree.
  - Generate code by visiting nodes in the parse tree in a given order.
  - Input: \( y := 3 \times x + z \)

```
parse tree
  :=
      id
      +
          *
              id
                      const
                              id
```

```
annotated parse tree
  :=
      id
      +
          *
              id
                      (y)
                              const
                                      id
                                              (z)
```

Compiler notes #4, Tsan-sheng Hsu, IIS
Each grammar symbol is associated with a set of attributes.

- **Synthesized attribute**: values computed from its children or associated with the meaning of the tokens.
- **Inherited attribute**: values computed from parent and/or siblings.

Format for writing syntax-directed definitions.

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L \rightarrow E$</td>
<td>$\text{print}(E.val)$</td>
</tr>
<tr>
<td>$E \rightarrow E_1 + T$</td>
<td>$E.val := E_1.val + T.val$</td>
</tr>
<tr>
<td>$E \rightarrow T$</td>
<td>$E.val := T.val$</td>
</tr>
<tr>
<td>$T \rightarrow T_1 \ast F$</td>
<td>$T.val := T_1.val \ast F.val$</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>$F \rightarrow \text{digit}$</td>
<td>$F.val := \text{digit.lexval}$</td>
</tr>
</tbody>
</table>

$E.val$ is one of the attributes of $E$.

To avoid confusion, recursively defined nonterminals are numbered on the LHS.
It is always possible to rewrite syntax-directed definitions using only synthesized attributes, but the one with inherited attributes is easier to understand.

- Use inherited attributes to keep track of the type of a list of variable declarations.
  ```latex
  \textgreater{} \textit{int} \ i, \ j
  ```

- Reconstruct the tree:
  ```latex
  \textgreater{} \ D \rightarrow TL \\
  \textgreater{} \ T \rightarrow \textit{int} | \textit{char} \\
  \textgreater{} \ L \rightarrow L, \textit{id} | \textit{id} \\
  ```
Attribute grammars (1/2)

- **Attribute grammar**: a grammar with syntax-directed definitions such that functions used cannot have **side effects**.
  - Side effect: change values of others not related to the return values of functions themselves.

- **$S$-attributed definition**: a syntax-directed definition that uses synthesized attributed only.
  - A parse tree can be represented using a directed graph.
  - A **post-order** traverse of the parse tree can properly evaluate grammars with $S$-attributed definitions.
  - **Bottom-up** evaluation.
Example of an $S$-attributed definition: $3 \times 5 + 4 \text{ return}$

$L$-attributed definition:

- Each attribute in each semantic rule for the production $A \rightarrow X_1 \cdots X_n$ is either a synthesized attribute or an inherited attribute $X_j$ depends only on the inherited attribute of $A$ and/or the attributes of $X_1, \ldots, X_{j-1}$.
- Independent of the evaluation order.
- Every $S$-attributed definition is an $L$-attributed definition.
Order of evaluation

- **Order of evaluating attributes is important.**
- **General rule for ordering:**
  - Dependency graph:
    - If attribute \( b \) needs attributes \( a \) and \( c \), then \( a \) and \( c \) must be evaluated before \( b \).
    - Represented as a directed graph without cycles.
    - Topologically order nodes in the dependency graph as \( n_1, n_2, \ldots, n_k \) such that there is no path from \( n_i \) to \( n_j \) with \( i > j \).
Orders for $L$-attributed definitions

- For grammars with $L$-attributed definitions, special evaluation algorithms must be designed.

- Bottom-up evaluation of $L$-attributed grammars.
  - Can handle all $LL(1)$ grammars and most $LR(1)$ grammars.
  - All translation actions are taken at the right end of the production.

- Key observation: when a bottom-up parser reduces by the production $A \rightarrow XY$, by removing $X$ and $Y$ from the top of the stack and replacing them by $A$, $X.s$ (the synthesized attribute of $X$) is on the top of the stack and thus can be used to compute $Y.in$ (the inherited attribute of $Y$).
Example for $L$-attributed definitions

- $D \rightarrow T \{L.in := T.type\} L$
- $T \rightarrow int \{T.type := integer\}$
- $T \rightarrow real \{T.type := real\}$
- $L \rightarrow \{L_1.in := L.in\} \ L_1, id \{\text{addtype}(id.entry, L.in)\}$
- $L \rightarrow id \{\text{addtype}(id.entry, L.in)\}$

Parsing and dependency graph:

<table>
<thead>
<tr>
<th>input</th>
<th>stack</th>
<th>production used</th>
</tr>
</thead>
<tbody>
<tr>
<td>int $p, q, r$</td>
<td>int $p, q, r$</td>
<td>$T \rightarrow int$</td>
</tr>
<tr>
<td>$p, q, r$</td>
<td>$T$</td>
<td></td>
</tr>
<tr>
<td>$p, q, r$</td>
<td>$T\ p$</td>
<td>$T \rightarrow int$</td>
</tr>
<tr>
<td>, $q, r$</td>
<td>$T\ L$</td>
<td>$L \rightarrow id$</td>
</tr>
<tr>
<td>$q, r$</td>
<td>$T\ L$</td>
<td></td>
</tr>
<tr>
<td>, $r$</td>
<td>$T\ L, q$</td>
<td></td>
</tr>
<tr>
<td>, $r$</td>
<td>$T\ L$</td>
<td>$L \rightarrow L, id$</td>
</tr>
<tr>
<td>$r$</td>
<td>$T\ L$</td>
<td></td>
</tr>
<tr>
<td>$T\ L, r$</td>
<td>$T\ L$</td>
<td></td>
</tr>
<tr>
<td>$T\ L, r$</td>
<td>$T\ L$</td>
<td>$L \rightarrow L, id$</td>
</tr>
<tr>
<td>$T\ L$</td>
<td>$D$</td>
<td>$D \rightarrow TL$</td>
</tr>
</tbody>
</table>

Compiler notes #4, Tsan-sheng Hsu, IIS
Implementation

Information contained in the stack can be used by replacing special markers to mark the production we are currently in.

<table>
<thead>
<tr>
<th>production</th>
<th>semantic rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \rightarrow aAC$</td>
<td>$C.in := A.s$</td>
</tr>
<tr>
<td>$S \rightarrow bABCD$</td>
<td>$C.in := A.s$</td>
</tr>
<tr>
<td>$C \rightarrow c$</td>
<td>$C.s := \ldots$</td>
</tr>
</tbody>
</table>

### Example 1:

Same rule for the first two productions. It is difficult to tell which one and to find the position of $A$ in the stack in each case.

<table>
<thead>
<tr>
<th>production</th>
<th>semantic rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \rightarrow aAC$</td>
<td>$C.in := A.s$</td>
</tr>
<tr>
<td>$S \rightarrow bABCD$</td>
<td>$M.in := A.s$; $C.in := M.s$</td>
</tr>
<tr>
<td>$C \rightarrow c$</td>
<td>$C.s := \ldots$</td>
</tr>
<tr>
<td>$M \rightarrow \epsilon$</td>
<td>$M.s := M.in$</td>
</tr>
</tbody>
</table>

### Example 2:

$A$ is always one place below in the stack.

Markers can also be used to perform error checking and other intermediate semantic actions.
Limitation

- Limitation of syntax-directed definitions: Without using global data to create side effects, some of the semantic actions cannot be performed.

- Example:
  - Checking whether a variable is defined before its usage.
  - Checking the type and storage address of a variable.
  - Checking whether a variable is used or not.

- Need to use a symbol table: global data to show side effects of semantic actions.

- YACC can be used to implement syntax-directed translations.

- Common approach:
  - A program with too many global variables is difficult to understand and maintain.
  - Restrict the usage of global variables to essential items and use them as objects.
    - Symbol table.
    - Labels for GOTO’s.
    - Forwarded declarations.
  - Use syntax-directed definitions as much as you can.