**Parser-Stack Implementation of Postfix SDTs**

Postfix SDTs can be implemented during LR parsing. Attribute of each grammar symbol can be put on the stack in place where they can be found during reduction, i.e., place attribute along with grammar symbol in record of stack itself.

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>state/grammar symbol</th>
<th>synthesized attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>X.a</td>
<td>Y.b</td>
<td>Z.c</td>
<td>top</td>
<td></td>
</tr>
</tbody>
</table>

Fig. parser stack with field for synthesised attributes

To implement SDT during LR parsing, add semantic stack parallel to the parsing stack: each symbol (terminal or non-terminal) on the parsing stack stores its value on the semantic stack. It holds terminals’ attributes and nonterminals’ translations. When the parse is finished, the semantic stack will hold just one value: the translation of the root non-terminal (which is the translation of the whole input). **Semantic Actions during Parsing**

- **when shifting**
  - push the value of the terminal on the semantic stack

- **when reducing**
  - pop k values from the semantic stack, where k is the number of symbols on production’s RHS
  - push the production’s value on the semantic stack

<table>
<thead>
<tr>
<th>Production</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L \rightarrow E\ n$</td>
<td>{ print ( stack [top - 1] val ); \ top = top - 1 ; }</td>
</tr>
<tr>
<td>$E \rightarrow E_1 \ + \ T$</td>
<td>{ stack [top - 2].val = stack[top - 2].val + stack[top].val; \ top = top - 2; }</td>
</tr>
<tr>
<td>$E \rightarrow T$</td>
<td></td>
</tr>
<tr>
<td>$T \rightarrow T_1 \ * \ F$</td>
<td>{ stack [top - 2].val = stack[top - 2].val x stack[top].val; \ top = top - 2; }</td>
</tr>
<tr>
<td>$T \rightarrow F$</td>
<td></td>
</tr>
<tr>
<td>$F \rightarrow (E)$</td>
<td>{ stack [top - 2].val = stack[top - 1].val; \ top = top - 1; }</td>
</tr>
<tr>
<td>$F \rightarrow \text{digit}$</td>
<td></td>
</tr>
</tbody>
</table>

The SDT for implementing the desk calculator on a bottom-up parsing stack is as above.
SDTs with Actions inside Productions

- Action can be placed at any position in the production body.
- Action is performed immediately after all symbols left to it are processed.
- Given B → X { a } Y, an action a is done after – we have recognized X (if X is a terminal), or – all terminals derived from X (if X is a nonterminal).
- If bottom-up parser is used, then action a is performed as soon as X appears on top of the stack.
- If top-down parser is used, then action a is performed – just before Y is expanded (if Y is nonterminal), or – check Y on input (if Y is a terminal).
- Any SDT can be implemented as follows:
  - Ignoring actions, parse input and produce parse tree.
  - Add additional children to node N for action in α, where A → α.
  - Perform preorder traversal of the tree, and as soon as a node labeled by an action is visited, perform that action.

SDT for infix-to-prefix translation during parsing

1) \( L \rightarrow E \ n \)
2) \( E \rightarrow \{ \text{print(‘+’); } \} E_1 + T \)
3) \( E \rightarrow T \)
4) \( T \rightarrow T_1 * F \{ \text{print(‘*’); } \} \)
5) \( T \rightarrow F \)
6) \( F \rightarrow ( \ E \ ) \)
7) \( F \rightarrow \text{digit} \{ \text{print(digit.lexval); } \} \)

Parse Tree with Actions Embedded

Eliminating Left Recursion from SDTs

Grammar with left recursion cannot be parsed using top-down parser. In case of SDT we treat action as terminal symbol in the production. Then we use the following rule of transforming grammar to non left-recursive form.
$A \rightarrow A\alpha \mid \beta$ can be transformed to $A \rightarrow \beta R, \quad R \rightarrow \alpha R \mid \epsilon$

In both forms, $A$ is defined by $\beta(\alpha)^*$

Example: Given $E \rightarrow E_1 + T \{\text{print('+');}\}$ and $E \rightarrow T$. Here $\alpha = + \ T \{\text{print('+');}\}$ and $\beta = T$. Modified grammar will be: $E \rightarrow T R, \quad R \rightarrow + T \{\text{print('+');}\} \quad R, \quad R \rightarrow \epsilon$.

More general example: $A \rightarrow A_1 Y \{ A.a=g(A_1.a, Y.y) \}$

$A \rightarrow X \{ A.a = f(X.x) \}$ can be rewritten as

$A \rightarrow X \{ R.i = f(X.x) \} R \{ A.a = R.s \}$

$R \rightarrow Y \{ R.i = g(R.i, Y.y) \} R_1 \{ R.s = R_1.s \}$

$R \rightarrow \epsilon \{ R.s = R.i \}$

SDTs for L-Attributed Definitions

- It is necessary that the underlying grammar can be parsed top-down
- Rules to modify L-attributed SDD to SDT
  - Embed the action for computing inherited attributes for a nonterminal $A$ immediately before $A$ in production body
  - Place the action for computing synthesized attribute for the head at the end of the body of production

Example 1: This example is motivated by languages for typesetting mathematical formulas. Eqn is an early example of such a language. It illustrates how the techniques of compiling can be used in language processing for applications other than programming languages.