I. Syntax Directed Translation

A syntax-directed translation is used to define the translation of a sequence of tokens to some other value, based on a CFG for the input. A syntax-directed translation is defined by associating a translation rule with each grammar rule.

A translation rule defines the translation of the left-hand-side nonterminal as a function of the right-hand-side nonterminals' translations, and the values of the right-hand-side terminals.

To compute the translation of a string, build the parse tree, and use the translation rules to compute the translation of each nonterminal in the tree, bottom-up; the translation of the string is the translation of the root nonterminal.

There is no restriction on the type of a translation; it can be a simple type like an integer, or a complex type list as an abstract-syntax tree.

Attributes and Semantic Rules

Example 1: Consider the Grammar

1. $E \rightarrow E_1 + T$
2. $| T$
3. $T \rightarrow T_1 * F$
4. $| F$
5. $F \rightarrow (E)$
6. $| \text{num}$

The corresponding semantic rules for the productions in the above grammar may be,

1. $E.v := E_1.v + T.v$
2. $E.v := T.v$
3. $T.v := T_1.v * F.v$
4. $T.v := F.v$
5. $F.v := E.v$
6. $F.v := \text{num}.v$ ("value" of token)

Example 2: Compute the type of an expression that includes both arithmetic and boolean operators. (The type is INT, BOOL, or ERROR.)

<table>
<thead>
<tr>
<th>CFG</th>
<th>Translation rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>$exp \rightarrow exp + exp$</td>
<td>if $((exp_2.\text{trans} == \text{INT}) \text{ and } (exp_3.\text{trans} == \text{INT})$ then $exp_1.\text{trans} = \text{INT}$ else $exp_1.\text{trans} = \text{ERROR}$</td>
</tr>
</tbody>
</table>
2. exp -> exp and exp  if ((exp₂.trans == BOOL) and (exp₃.trans == BOOL))
              then exp₁.trans = BOOL
              else exp₁.trans = ERROR

3. exp -> exp == exp  if ((exp₂.trans == exp₃.trans) and (exp₂.trans != ERROR))
              then exp₁.trans = BOOL
              else exp₁.trans = ERROR

4. exp -> true        exp.trans = BOOL
5. exp -> false       exp.trans = BOOL
6. exp -> int         exp.trans = INT
7. exp -> ( exp )     exp₁.trans = exp₂.trans

Input: (2 + 2) == 4

Annotated Parse Tree

exp (BOOL)
  /\   
  /   \ 
(IN) exp == exp (INT)
  /\   
  /   \ 
  / \  4
  / \ 
  /   \ 
  ( exp )
(IN)
  /\ 
  / \ 
  (INT) exp + exp (INT)
    | | 
    2 2

Associating attributes with grammar symbols, and semantic rules with productions give
us a syntax directed definition.

II. Translation Schemes

If we embed program fragments called semantic actions within the RHS’s of
productions, we will have a translation scheme.
The translation scheme explicitly shows the order of evaluation of the actions – (assuming we can imply the order in which the parse tree will be covered)

**Example 3:** Here is a scheme which concatenates simply by writing (appending) to a file

**Implied Traversal order**

- We must “visit” child nodes before we can visit an interior node
- Execute the action when we “visit”

```plaintext
1. E \rightarrow E + T { print ‘+’ }
2. \mid T
3. T \rightarrow T * F { print ‘*’ }
4. \mid F
5. F \rightarrow ( E )
6. \mid num { print num }
```

In syntax-directed translation, we maintain a *semantic stack* along with the parse stack. The basic rules governing the semantic stack are as follows:

- When a terminal is parsed, its *attribute* is pushed on the stack. For constructing ASTs, the token itself suffices as the attribute for terminals.
- When a non-terminal is parsed, its translation is pushed on the stack. In our case, the translation of a non-terminal will be an AST node.
- Values are pushed onto the semantic stack (and popped off) by adding *actions* to the grammar rules. The action for a rule must:
  - Pop the translations of all right-hand-side nonterminals.
  - Compute and push the translation of the left-hand-side nonterminal.
- The actions themselves are represented by *action numbers*, which become part of the right-hand sides of the grammar rules. They are pushed onto the (normal) stack along with the terminal and nonterminal symbols. When an action number is the top-of-stack symbol, it is popped and the action is carried out.

So what actually happens is that the action for a grammar rule $X \rightarrow Y_1 \ Y_2 \ ... \ Y_n$ is pushed onto the (normal) stack when the derivation step $X \rightarrow Y_1 \ Y_2 \ ... \ Y_n$ is made, but the action is not actually performed until complete derivations for all of the $Y$’s have been carried out.
At the completion of a successful parse, the semantic stack should have one element, the **translation of the start symbol**. When doing bottom-up (LR) parsing, maintaining the semantic stack is straightforward; we essentially “mirror” the transformations done to the parse stack. When we shift a terminal, we push its attribute on the semantic stack. When we reduce to a non-terminal, we pop the items on the semantic stack corresponding to the items popped from the parse stack, and then push the computed translation for the non-terminal on the semantic stack.

Unfortunately, when doing top-down parsing (LL(1) or recursive-descent), performing syntax directed translation is not so easy, since the translation is performed bottom-up. We solve this problem by adding *actions* to our grammar, specifying how translation is to be performed for each production. In LL(1) parsing, these actions (more specifically, numbers representing the actions) are pushed on the parse stack along with the corresponding production, and then executed when popped. These actions are responsible for popping all the items on the semantic stack corresponding to terminals and non-terminals on the right-hand side of the production, and then pushing the computed translation.

**Example 4 : Counting Parentheses**

For example, consider the following syntax-directed translation for the language of balanced parentheses and square brackets. The translation of a string in the language is the number of parenthesis pairs in the string.

<table>
<thead>
<tr>
<th>CFG</th>
<th>Translation Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. exp → epsilon</td>
<td>exp.trans = 0</td>
</tr>
<tr>
<td>2. → ( exp )</td>
<td>exp₁.trans = exp₂.trans + 1</td>
</tr>
<tr>
<td>3. → [ exp ]</td>
<td>exp₁.trans = exp₂.trans</td>
</tr>
</tbody>
</table>

The first step is to replace the translation rules with **translation actions**. Each action must:
- Pop all right-hand-side nonterminals' translations from the semantic stack.
- Compute and push the left-hand-side nonterminal's translation.

Here are the translation actions:

<table>
<thead>
<tr>
<th>CFG</th>
<th>Translation Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. exp → epsilon</td>
<td>push(0);</td>
</tr>
<tr>
<td>2. → ( exp )</td>
<td>exp2Trans = pop(); push( exp2Trans + 1 );</td>
</tr>
<tr>
<td>3. → [ exp ]</td>
<td>exp2Trans = pop(); push( exp2Trans );</td>
</tr>
</tbody>
</table>
Next, each action is represented by a unique action number, and those action numbers become part of the grammar rules:

**CFG with Actions**

1. $\text{exp} \rightarrow \epsilon$ #1
2. $\rightarrow (\text{exp})$ #2
3. $\rightarrow [\text{exp}]$ #3

#1: push(0);

#2: exp2Trans = pop(); push( exp2Trans + 1 );

#3: exp2Trans = pop(); push( exp2Trans );

## III. Semantic Actions

There are 3 types of semantic actions necessary to give meaning to the syntactic structure. These actions are:

1. Setting attributes in the symbol table, e.g. the number of bits in a BIT variable, the dimensions of an ARRAY variable and more commonly setting type bits, i.e. real, integer, external, label, octal, etc.

2. Writing intermediate code, such as 4-tuples to indicate an arithmetic operation, a label, a jump, a conditional test, etc.

3. Passing of semantic information through the semantic stack.

**Note**: The syntax and semantics of a language are not orthogonal. It is not always possible to create the syntax and then later somehow define the semantics. For example, consider the following set of productions:

Suppose in a language, the statement for declaring variables is

**ABC, AAA, DFG INTEGER**

and the following syntax was used to "parse" this string

1. $\text{decl} \rightarrow \text{var}$
2. $\text{decl} \rightarrow \text{decl} , \text{var}$
3. $\text{declstat} \rightarrow \text{decl integer}$
Then when the type of the declared variables is determined, i.e. production 3 is recognized, *all of the preceding variables have been lost* and their attributes in the symbol table cannot be set!

**Example 5**: consider the following production for declarations,

\[
\text{decstat} \rightarrow \text{VECTOR var ELEMENTS const}
\]

This statement in the language would translate into something like

**VECTOR AQZ ELEMENTS 50**

Then supposing we were using a Bottom-Up Parser, the syntax and semantics stacks would look something like the following:

BEGIN REDUCTION

<table>
<thead>
<tr>
<th>SYNTAX</th>
<th>SEMANTICS</th>
<th>SYNTAX</th>
<th>SEMANTICS</th>
</tr>
</thead>
<tbody>
<tr>
<td>con</td>
<td>157</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ELEMENTS</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>var</td>
<td>113</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VECTOR</td>
<td>0</td>
<td>decstat</td>
<td>4 157</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td></td>
<td>-</td>
</tr>
</tbody>
</table>

**Explanation**

In the BEFORE stack the 113 is the symbol table location of AQZ, and the 157 is the symbol table location of 50. When reducing this production, the symbol table entry of AQZ needs to have its attributes set to vector, which in this example is represented by the numeric code 4, and the first dimension set to the contents of the symbol table entry at location 157 which contains 50. At this instant in time the symbol table entry at location 113 has the symbol name of AQZ; the type is vector; the number of elements is 50.

**Example 6**: Now consider the following production for executable statements,

\[
factor \rightarrow \text{factor} \ast \text{primary}
\]

When this reduction is made, the stack will look something like,
The partial string

...... Y / X * ( I + 2 ) .......

could have been the string that represents the stack where Y / X is syntactically the factor and ( I + 2 ) is the primary.

The semantics for the above production might be

1. If the types of the two operands are incompatible, then replace the incompatible operand with the next intermediate result name of the compatible type and write the 4-tuple (IR$_k$, CONVIR, S$_i$, -). This step may be skipped if the language does not support automatic type upgrading.

2. When both operands are compatible, generate the tuple

   (IR$_{k+1}$, RMULT, S$_i$, S$_{i-2}$)

   where IR$_{k+1}$ indicates the next real intermediate result name.

3. Finally, place the new intermediate result name in the semantic stack at S$_{i-2}$. Assume, S$_i$ is the top of the semantic stack.

Note: Both example 5 and example 6 are for Bottom-Up Parsers