Syntax Directed Translation

Parsing is the act of checking that a sentence can be produced by a given grammar. However, in a compiler, one must do more than this. Eventually, code must be produced (i.e., a translation) and one way to do this is by defining semantic actions for various parts of the grammar. This is called *syntax directed translation*.

Syntax directed translation consists of a context-free grammar, a set of attributes for each grammar symbol and a collection of semantic rules associated with each production. The attributes are either *synthesized* or *inherited*. A simple way to envision this is to think of each grammar symbol as a record in the parse tree, where attributes are a name of a field in the record.

Attributes are computed by the semantic actions associated with the rule. If an attribute is computed from only child nodes, it is *synthesized*. If it computed from parent and/or sibling nodes, it is *inherited*. We call a parse tree with the attributes an *annotated parse* *tree* and the act of adding the attributes *decorating* or *annotating* the parse tree.

Infix to Postfix Example

$$E \rightarrow E_1 + T \; \{E = E_1.val \,||\, T.val \,||\, +\}$$

$$E \rightarrow E_1 - T \; \{E = E_1.val \,||\, T.val \,||\, -\}$$

$$E \rightarrow T \; \{E.val = T.val\}$$

$$T \rightarrow 0| \ldots |9 \; \{T.val = \text{number}\}$$

(The $||$ operator is concatenation. It is assumed the *val* attribute is actually a string)

Using the above scheme, we can translate from infix to postfix for simple additive expressions. If you draw out the parse tree for some expression and annotate the parse tree, you can translate

```
3 + 2 - 4
```

into

```
3 2 + 4 -
```

by evaluating the semantic actions from the bottom-up.

It is also possible to do the same thing in a slightly different manner.

$E \rightarrow E_1 + T$ {`print ('+');`}

$E \rightarrow E_1 - T$ {`print ('-');`}

$E \rightarrow T$

$T \rightarrow 0 | \ldots | 9$ {`print ('%d', number);`}

The correct translation can now be achieved by doing a postorder traversal (a.k.a. depth-first order) of the annotated parse tree.

Notice there are various ways in which to make the order of the semantic actions specific. For bottom-up parsing, the first technique is more suitable. It is known as a *simple translation scheme* because the output is merely the concatenation of the attributes for the grammar symbols on the right side of the production.

Synthesized and Inherited Attributes

Synthesized Attributes are those that can be determined in terms of the attributes of child nodes. Thus, for a production $A \to \alpha$, we can determine the values for attributes in A by only looking at attributes in α. A syntax directed translation with only synthesized attributes is called an *S-attributed definition*. S-attributed definitions are easily implemented by semantic actions in an LR parser since both evaluate from the bottom up.

Inherited Attributes are those that are determined in terms of attributes of sibling and/or parent nodes. If X is a grammar symbol with inherited attributes, then for some production $A \to \alpha X \beta$, attributes in X are determined by a combination of attributes in A, α or β.

Syntax-Directed Definitions

Every grammar rule of the form $A \to \alpha$ has a set of semantic actions associated with it. They are of the form $b = f(c_1, c_2, \ldots, c_k)$ where:

- b is a synthesized attribute of A and c_i are attributes of grammar symbols of the production
- b is an inherited attribute of one of the grammar symbols in α and c_i are attributes of grammar symbols in the production.

In either case, we say that b *depends* on $c_1, c_2, \ldots, c_k$.

Terminals are considered to only have synthesized attributes whose values are usually supplied by the lexical analyzer. The start symbol is assumed to have only synthesized attributes.

Dependency Graphs

We can create a dependency graph for the attributes for a syntax directed translation by drawing a graph with a directed edge from c_i to b when attribute b depends on c_i. If a semantic action only creates a side effect (for example, a print statement or a function call), then we can introduce a dummy attribute for that action.

Attributes must be evaluated in the order defined by the dependency graph. If the dependency graph for some parse tree of the underlying grammar has a cycle, the syntax-directed definition is said to be circular. If the syntax-directed definition is circular, the attributes cannot be evaluated. However, there is no efficient algorithm to test for circularity in syntax-directed definitions.

After creating the dependency graph, we can perform a *topological sort* on the graph (assuming it has no cycles). Note that a dependency graph is a directed, acyclic graph. Thus, we can impose an ordering on the nodes in the graph such that for any edge (m_i, m_j), $i < j$. After creating this graph, we can list, in order, the evaluation of the semantic rules of the entire syntax-directed definition.

1. Create the parse tree

2. Create the dependency graph for the parse tree

3. Do a topological sort of the dependency graph

Syntax Trees

A *syntax tree* is a condensed form of parse tree. It used to separate translation from parsing. Usually, syntax are more convenient to deal with than full parse trees.

A node in a syntax tree is usually an operator or language construct with the arguments as children. For example:

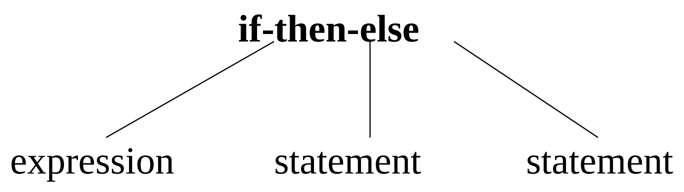

Figure 1: Syntax tree node.

The noticeable difference between a syntax tree and a parse tree is that "useless" nodes are removed.

Example:

Production	Semantic Rules
$E \rightarrow E_1 + T$	`E.nptr = mknode ('+', E1.nptr, T.nptr)`
$E \rightarrow E_1 - T$	`E.nptr = mknode ('-', E1.nptr, T.nptr)`
$E \rightarrow T$	`E.nptr = T.nptr`
$T \rightarrow T_1 * F$	`T.nptr = mknode ('*', T1.nptr, F.nptr)`
$T \rightarrow T_1 / F$	`T.nptr = mknode ('/', T1.nptr, F.nptr)`
$T \rightarrow F$	`T.nptr = F.nptr`
$T \rightarrow (E)$	`T.nptr = E.nptr`
$F \rightarrow \mathbf{id}$	`F.nptr = mkleaf (id, id.entry)`
$F \rightarrow \mathbf{num}$	`F.nptr = mkleaf (num, num.val)`

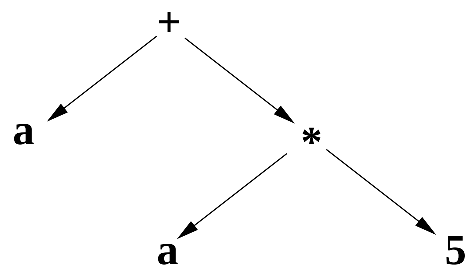

Figure 2: Syntax tree for `a + a * 5`.

Directed Acyclic Graphs (DAGs)

We can use a DAG for creating syntax trees as well. The advantage to DAGs is that they avoid redundant subtrees. Essentially, any common nodes are merged into a single node. This is done by altering the procedures `mknode` and `mkleaf` to first check to see if the node already exists.

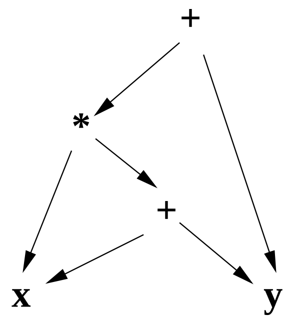

Figure 3: DAG for `x * (x + y) + y`.

While DAGs avoid redundant code, they can be inefficient and problematic later on if there are side effects that change values at some point.

In order to implement a DAG, usually the nodes are stored in an array and searched when a new node is to be created. A signature can be made through a (operator, expr, expr) triple, thus allowing the value to be hashed. This creates a much faster method for searching as opposed to simply scanning a large array with all the nodes. Plus, it makes addition and removal of nodes more efficient.