9 Synthesis 1 - Semantics

So far we have concentrated on how to analyse the form of a program and to detect syntactic errors. The question of semantics has been ignored. Once we have parsed the program, however, we wish to associate meaning with the phrases. This means associating semantic definitions with syntactic constructions.

Most programming language definitions contain formal descriptions of the syntax. Some, such as Algol 68, introduce a second layer, sometimes described as "long range syntax", to describe type checking and declaration constraints. This concept is not universally accepted, however.

There are three approaches which offer means of associating semantic actions with syntactic structures. The first two are variants of attribute grammars. The third, which is very new, is operational semantics. We shall concentrate on attribute grammars. Operational semantics is presented in CS3 Language Semantics and Implementation.

The two variants of attribute grammar are:

- syntax directed definitions
- translation schemes

Attribute grammars offer the possibility of generating code generators automatically, for particular machines.

Syntax directed definitions

input string $\rightarrow$ parse tree $\rightarrow$ dependency graph $\rightarrow$ evaluation order for semantic rules

A syntax directed definition takes a context free grammar and associates attributes with each symbol. Attributes are defined as synthesised or inherited.

When we build the parse tree, the nodes representing symbols have fields to hold the attributes for that symbol. Such attributes may be strings, values, addresses or whatever is required.

For each attribute there is a semantic rule defining its evaluation.

A synthesised attribute is calculated from attributes of the children of that node.

An inherited attribute is calculated from attributes of the siblings or parent of that node.

These semantic rules create dependencies amongst nodes. These can be represented as a dependency graph and from this graph an order of evaluation can be calculated. Syntax directed definitions do not define in advance the order of evaluation.

The semantic rules are associated with production rules. These rules define the value of attributes of the left hand symbol in each production as a function of attributes of symbols on the right hand side or define the value of an attribute of one of the symbols on the right hand side in terms of attributes of the left hand symbol and attributes of the other right hand symbols. The first case defines a synthesised attribute, the latter an inherited attribute.

In each case the attribute defined depends on those which define it.

Strictly speaking, in an attribute grammar, these rules must not produce side effects. Some syntax directed definitions have attributes which are procedures, which produce side effects and do not generate interesting values. These are not really pure attribute grammars.

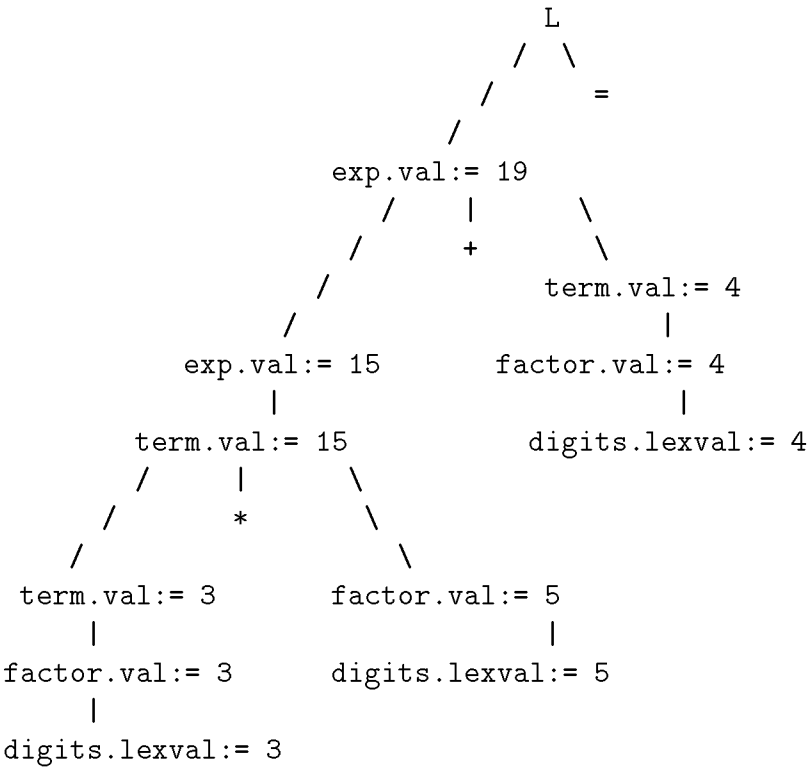
Example: Pocket calculator

Production			Semantic rules
L	→	exp =	print(E . val) ! not a value generator
exp	→	exp + term	exp.val:= exp1.val + term.val
exp	→	term	exp.val:=term.val
term	→	term ∗ factor	term.val:= term1.val X factor.val
term	→	factor	term.val:=factor.val
factor	→	(exp)	factor.val:=exp.val
factor	→	digits	factor.val:= digits.lexval

Note that terminals, like digits, can only have synthesised attributes. These are usually lexical values.

Equally, the start symbol normally has no inherited attributes.

The following annotated parse tree shows this for `3 * 5 + 4 =`



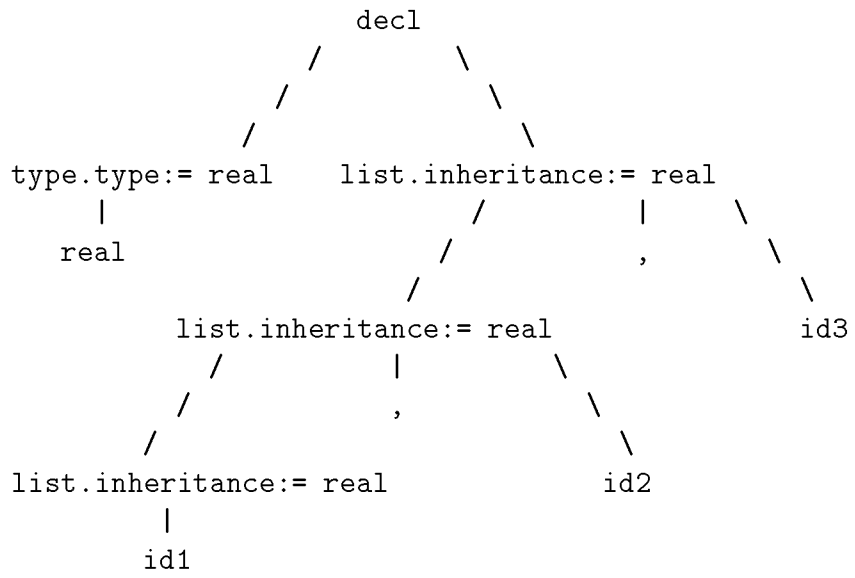
In each case the value of the attribute of each non-terminal is synthesised from the values of its branch's or branches' attributes.

Inherited attributes - types

Consider the following syntax directed definition:

Production	Semantic Rules
decl → typelist	list.inheritance:= type.type
type → int	type.type:= integer
type → real	type.type:= real
list → list1 , id	list1.inheritance:= list.inheritance
	addtype(id.entry, list.inheritance)
list → id	addtype(id.entry, list.inheritance)

The annotated parse tree for `real, id1, id2, id3` follows.



The procedure `addtype` instructs the compiler to add the appropriate type information to the symbol table entry for each `id`. This type information is first inherited by the list from the type in

decl → typelist

It is then inherited by each sub-list and given to each `id` in the productions

list → list1 , id

and

list → id.

Dependency graphs

To generate a dependency graph we assume that procedure calls are actually functions whose values are assigned to dummy attributes of nodes. This allows us to treat them consistently with real attributes when determining the order of execution of the semantic rules.

To construct a dependency graph from an annotated parse tree we

1. create a node for each attribute of each parse tree node.
2. for each semantic rule, construct an edge from each node of an attribute on the right hand side of an assignment to its corresponding left hand side's node.

Thus the arrow on the edge says that the node it points towards depends on the node it leaves.

Thus, our type declaration parse tree gives us

Using this graph we can now determine the evaluation order by performing a topological sort. This orders the nodes so that if there is an edge $n_1 \rightarrow n_2$ then n_1 has a lower numbering than n_2. As long as this is true, any ordering is valid in evaluating this tree. This is so because it must be true that all right hand side attributes in semantic rules are evaluated before their corresponding left hand sides.

Translation schemes

Translation schemes also define attributes for parse tree nodes. Unlike syntax directed translations, they specify the semantic actions within the right hand side of productions. These are, effectively, treated as terminals in the parse tree, corresponding to the empty string between their neighbouring real symbols.

A simple example shows this for grammar.

$$
\begin{array}{rcl}
\mathsf{exp} & \rightarrow & \mathsf{term}\ \mathsf{rest} \\
\mathsf{rest} & \rightarrow & \mathsf{addop}\ \mathsf{term}\ \{\text{print(addop.lexeme)}\}\ \mathsf{rest} \mid \square \\
\mathsf{term} & \rightarrow & \mathsf{num}\{\text{print(num.val)}\}
\end{array}
$$

The string `9 - 5 + 2` parses to

Evaluating the tree left to right, depth first gives the postfix form 9 5 - 2 +.

Restrictions Evaluation which only involves synthesised attributes creates few problems in ensuring the correct order of evaluation. Thus

production	semantic rule
term → term1 * factor	term.val:= term1.val X factor.val

can be written with the rule at the end of the production.

term → term1 * factor	term.val:= term1.val X factor.val

Where there are also inherited attributes the ordering is more difficult. The following rules must be applied.

1. An inherited attribute for a symbol on the right hand side must be calculated in an action before that symbol.

2. An action must not refer to a synthesised attribute of a symbol to its right.

3. A synthesised attribute for the left hand non-terminal in a production can only be computed after all the attributes it references. These computations can usually go at the right hand end of the right hand side of that production.