Summary: Semantic Analysis

1 Basic Concepts

When SA is performed: Semantic Analysis may be performed:

- **In a two-pass compiler**: after syntactic analysis is finished, the semantic analyser
  if called with the syntactic tree as input.
- **In a one-pass compiler**: semantic analysis is performed on each node of the parse
  tree as it is constructed. In bottom up parsing, this means that as each production
  rule is reduced, the semantic actions associated with the rule are applied.

What does SA do? Semantic Analysis is used to check the global integrity of a program.
Syntactic analysis can only check that a series of symbols are syntactically correct.
Semantic analysis can compare information in one part of a parse tree to that in another
part (e.g., compare reference to variable agrees with its declaration, or that parameters to
a function call match the function definition).
Semantic Analysis is used for the following:

- Maintaining the Symbol Table for each block;
- Reporting compile-time errors in the code (except syntactic errors, which are
  caught by syntactic analysis)
- Generating the object code (e.g., assembler or intermediate code)

2 Attribute Grammars

Semantic Analysis is performed via an extension over the usual context free grammar
(CFG). The extended grammar is called an “attribute grammar”. An Attribute Grammar is a
CFG grammar with three extensions:

1. **Attributes on Symbols**: Each grammar symbol S, terminal or nonterminal,
is specified to have various attributes. Each attribute is specified both with a
name (e.g., ‘type’, ‘value’, and also a type, restricting the range of fillers the
attribute can take. E.g. Expr.type, ID.symbol, etc.

2. **Attribute evaluation rules**: Each production rule in the grammar can have
associated with it a number of ‘semantic rules’, each of which specifies how
an attribute of one symbol can be calculated from the attributes of other
symbols in the production. E.g.,

   Decl :- Mode IDList       { IDList.type = Mode.type }

3. **Indexing of grammar symbols**: The same grammar symbol can occur more
than once in a CFG rule. To allow a semantic rule to distinguish between
each occurrence of a grammar symbol in the production rule, the
occurrences are indexed E.g.,

   Expr₁ :- Expr₂ + Expr₃    {Expr₁.type = Expr₂.type }
3 Synthesised and Inherited Attributes

Attributes can be of two types:

- **Synthesised attributes**: The attributes of the unit are calculated by looking at the attributes of CHILDREN of that unit (in other words, the attributes of the LHS of a production rule are calculated by looking at the attributes of the symbols on the RHS of the production rule).
  - Example: value attribute in the example or arithmetic expressions:
    - \[ E \rightarrow E_1 + E_2 \{ E.\text{val} = E_1.\text{val} + E_2.\text{val} \} \]

- **Inherited attributes**: Attributes of the unit are calculated using the attributes of the PARENT (or sister) units in the tree.
  - An example is the type attribute, in the second example:
    - \[ D \rightarrow T L \{ D.\text{type} = T.\text{type}; L.\text{type} = D.\text{type} \} \]

Inheritance can be from the parent node, e.g.,
- \[ D \rightarrow T L \{ L.\text{type} = D.\text{type} \} \]

Alternatively, inheritance can be from a sister unit:
- \[ D \rightarrow T L \{ L.\text{type} = T.\text{type} \} \]

A evaluation rule for a synthesized attribute will always have the LHS of the production on the LHS of the evaluation rule, e.g.,
- \[ \text{Expr} :\text{-} \text{Expr1} \ ' + ' \text{Expr2} \{ \text{Expr}.\text{val} = \text{Expr1}.\text{val} + \text{Expr2}.\text{val} \} \]

A evaluation rule for a inherited attribute will always a symbol from the RHS of the production on the LHS of the evaluation rule, e.g.,
- \[ \text{L} :\text{-} \text{L1}, \text{id} \{ \text{L1}.\text{type} = \text{L}.\text{type} \} \]

In real grammars, a given attribute might be mixed, i.e., in some semantic rules the grammar symbol which is having an attribute assigned is from the left of the CFG rule, and in other rules, it is from the right of the CFG rule.

4 Strict and Extended Attribute Grammars

Attribute evaluation rules have an abstract form like:
- \[ X.\text{attrib} := f(Y.\text{attrib}, Z.\text{attrib}) \]
...where \( f() \) is a function over attributes.

In a **strict attribute grammar**, the functions on the RHS of attribute evaluation rules should not have 'side effects', i.e., they should not change the structure of the parse tree, nor the attributes of any symbols (the only way to change an attribute of a symbol is by being on the LHS of a semantic rule.).

An **extended attribute grammar** allows side effects:

- The functions \( f() \) can change the values of other attributes;
- They can change values of global data structures (e.g., add an entry to a symbol table);
- They can re-structure the parse tree.

## 5 Applying Semantic Rules

An annotated parse tree is a syntactic parse tree with all attributes shown, along with their values:

![Annotated Parse Tree Diagram]

The semantic rules can be applied in various orders. The approach to ordering these rules can be divided between:

- **Ordering with a one-pass compiler**: Semantic rules are applied as each CFG rule is applied (the rule is used to replace a set of symbols on the stack corresponding to the RHS of the CFG rule to the LHS of the rule, i.e., a reduce operation).

- **Ordering with a two-pass compiler**: semantic rules are applied after syntactic parsing is complete. Two main methods are used:
  - **Recursive Descent Approach**: We start with the top of the tree, and:
    1. Evaluate all semantic rules for this node where the attributes on the RHS of the semantic rule are known. Some rules cannot be applied at this point because the values of attributes are not yet known.
    2. Call this method on each of the children of this node to evaluate attributes.
3. After children processed, try to evaluate unresolved semantic rules, as the attributes of children may now be available.
(Where attributes depend on other attributes to their right in the tree, this process may need to be repeated several times to resolve all attributes)

- **Dependency Graph Approach**: build a directed graph showing the dependency between all attributes:
  1. For each semantic rule:
    - Draw the attributes on the RHS of the semantic rule with an arrow pointing to the attribute on the LHS of the rule, e.g.,
      
      \[
      \begin{align*}
      \text{Given: } & \quad A.v = B.v + C.v \\
      \text{B.v} & \quad \rightarrow \quad A.v \\
      \text{C.v} & \quad \rightarrow \quad A.v
      \end{align*}
      \]
    
    - Where an attribute on the RHS of the rule was previously on the LHS, merge the rules, e.g.,
      
      \[
      \begin{align*}
      \text{Adding: } & \quad A.u = A.v \\
      \text{B.v} & \quad \rightarrow \quad A.v \quad \rightarrow \quad A.u \\
      \text{C.v} & \quad \rightarrow \quad A.v
      \end{align*}
      \]
    
    - Where the attribute on the LHS was previously on the RHS of a rule, merge, e.g.,
      
      \[
      \begin{align*}
      \text{Adding: } & \quad C.v = D.v \\
      \text{B.v} & \quad \rightarrow \quad A.v \quad \rightarrow \quad A.u \\
      \text{D.v} & \quad \rightarrow \quad C.v
      \end{align*}
      \]
    
    - etc.
  2. Once the graph is constructed:
    - If there is a cycle in the graph, the semantic attributes cannot be calculated, stop
    - Otherwise, produce a **topological sort** of the graph (any one of the possible orderings of the graph where dependent attributes are listed after the nodes they depend on), e.g. from the graph above, we have 3 topological sorts:
      
      \[
      \begin{align*}
      \text{D.v} & \quad C.v \quad B.v \quad A.v \quad A.u \\
      \text{D.v} & \quad B.v \quad C.v \quad A.v \quad A.u \\
      \text{B.v} & \quad D.v \quad C.v \quad A.v \quad A.u
      \end{align*}
      \]
If asked to show attribute dependency, you can show attributes attached to the nodes in the parse tree, e.g.,

```
Number  value
  base  Digit_Seq  value  base  Base_Tag
      base  digit  value
```

By convention, inherited attributes are shown on the left of the CFG symbol, and synthesised attributes on the right (e.g., down on the left, and up on the right). Where an attribute is mixed (sometimes synthesised, sometimes inherited), consider its most common use.

## 6 Restricted Attribute Grammars for 1-pass compilers

In a single-pass compiler, where semantic rules are applied as grammar rules are applied, we cannot guarantee all attributes will be available at the time of rule application. For this reason, restricted attribute grammars have been developed for this use.

For use in bottom-up parsing:

1. **S-Attributed grammar**: only allows synthesis of attribute values, i.e., attributes are assigned to the LHS of the CFG rule from attributes of the constituents of this LHS. This means that, at the point of applying a reduce operation, all attribute values are on hand to calculate the attributes of the parent.

2. **L-Attributed grammar**: allows both synthesised and inherited attributes with one restriction: a node’s attributes cannot inherit from the attributes of nodes to its right. I.e.

```
P
  ↓
 C1  C2  C3
```

More formally, given a parse tree node C, the attributes of C can be derived from:

i) **synthesized** attributes: attributes of children of C;

ii) **inherited** attributes where the inherited attribute depends only on:

   a) attributes of the **other child nodes** to the left of C;
   
   b) attributes **inherited** from the parent node of C;

Note: an s-attributed grammar is a type of L-Attributed grammar since it meets these restrictions.
Note: If one uses an L-attributed grammar in a two-pass compiler, the recursive descent approach to resolve attributes is guaranteed to terminate in a single cycle.

**Use of L-Attributed Grammars in Top-down parsing:** L-attributed grammars are well-suited to top down parsing. TO BE DISCUSSED AS PART OF LL PARSING.

**Use of L-Attributed Grammars in Bottom-up parsing**
With careful writing, L-Attributed grammars can be used in a single-pass bottom-up parser.
- Synthesised attributes are not a problem because children are constructed before the parent unit.
- Inheritance from the left is not a problem because the leftmost constituents are recognised (and their attributes resolved) before those more to the right.
- The only problem is with inheritance from above.

**Embedding Actions in CFG Rules**
So far, we have considered that semantic actions are only applied AFTER the CFG rule has been totally recognised (after the reduction). Allowing semantic actions to be placed BETWEEN THE SYMBOLS of the CFG rule would mean that the semantic actions could be applied before the rest of the rule has been recognised.

For instance, an abstract rule could appear as:

\[ A :: M1 \text{ $action1$ } M2 \text{ $action2$ } M3 \text{ $action3$} \]

For example:

\[ E :: T \{ R.i := T.val \} \text{ R } \{ E.val := R.s \} \]

Often Presented as:

\[ E :: T \{ R.i := T.val \} \]

\[ R \{ E.val := R.s \} \]

This would mean, in an LR parser, if we have just recognised a T element, we then expect a following R element, and we can evaluate the ‘i’ attribute of this element in advance. This would mean that, at the point of recognising an R element, we would know the value of R.i, and thus the constituents of R could use this attribute to calculate their attributes. Thus, in a restricted manner, inheritance from above is allowed by this mechanism in a single-pass bottom-up analyser.

**Problem with Embedded Actions in LR parsing**
In an LR parser, a given parsing state may be processing several alternative productions, e.g.,

\[ S_8 \]

\[ E :: ( E . ) \]

\[ E :: ( E . + T ) \]

If the current parse stack has symbols:

Stack: ( E )
...then it is not clear if we are working on $E : (E+T)$ or $E : (E)$. Only when we ‘reduce’ is the decision made as to which rule is intended. **THUS**, we cannot perform actions placed between RHS symbols!

A solution to the problem is to use lambdas. Rather than:

$$ E : \quad T \quad \{R.i := T.val\} $$
$$ R \quad \{E.val := R.s\} $$

...we use instead:

$$ E : \quad T \quad X \quad R \quad \{E.val := R.s\} $$
$$ X : \quad \lambda \quad \{R.i := T.val\} $$

The semantic action is no longer between symbols of the CFG rule. Rather, it is attached to a lambda rule. When the parser decides to reduce $X$, the rule is fired. The grammar is written such that the next input symbol will resolve whether to recognise an $X$, or to do some other operation. So the semantic action is no longer associated with an ambiguous state.

Note however that the action on the lambda rule makes reference to symbols from another rule:

$$ X : \quad \lambda \quad \{R.i := T.val\} $$

Yacc is written to take account of the context of occurrence of lambda productions, and interpret the symbols in the context of grammar rule where $X$ occurs on the RHS (e.g. $E : \quad T \quad X \quad R$)