Syntax-directed translation → the source language translation is completely driven by the parser
- The parsing process and parse trees/AST used to direct semantic analysis and the translation of the source program
- Separate phase of a compiler or grammar augmented with information to control the semantic analysis and translation (attribute grammars)

Attribute grammars → associate attributes with each grammar symbol
- An attribute has a name and an associated value: string, number, type, memory location, register — whatever information we need.
- Examples
  - Attributes for a variable include type (as declared, useful later in type-checking)
  - An integer constant will have an attribute value (used later to generate code)
- With each grammar rule we also give semantic rules or actions, describing how to compute the attribute values associated with each grammar symbol in the rule
  - An attribute value for a parse node may depend on information from its children nodes, its siblings, and its parent

Grammar

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Action(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>⟨int⟩ ::= ⟨digit⟩</td>
<td>{ ⟨int⟩.value = ⟨digit⟩.value; }</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>⟨digit⟩ ::= 0</td>
<td>{ ⟨digit⟩.value = 0; }</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Attributes are computed during the construction of the parse tree and are typically included in the node objects of that tree
- Two general classes of attributes:
  - Synthesized: passed up in the parse tree
  - Inherited: passed down the parse tree

- Synthesized attributes: the left hand-side attribute is computed from the right hand-side attributes
  \[
  X ::= Y_1 Y_2 ... Y_n \\
  X.a = f(Y_1.a, Y_2.a, ..., Y_n.a)
  \]

- Inherited attributes: the right hand-side attributes are derived from the left hand-side attributes or other right hand-side attributes
  \[
  X ::= Y_1 Y_2 ... Y_n \\
  Y_k.a = f(X.a, Y_1.a, Y_2.a, ..., Y_{k-1}.a, Y_{k+1}.a, ..., Y_n.a)
  \]
  - Used for passing information about the context to nodes further down the tree
A given style of parsing favors attribute flow in one direction
- Top-down parsing deals trivially with inherited attributes
- Bottom-up parsing deals trivially with synthesized attributes
- The other direction is handled using other techniques
- For example, a symbol table is often used to pass attributed back and forth irrespective of the direction favored by any particular parsing method

Typically handling of attributes: associate with each symbol either member variables in the AST node structure or some sort of structure (e.g., list) with all the necessary attributes
- If we have a list then we store it as a member variable in each node structure
- Associate code to the processing of each nonterminal to carry on the attribute computations
- Also need some convention for referring to individual symbols in a rule while defining the associated action
- Typical convention in compiler generators: $\$$ to refer to the left hand side and $i$ to refer to the $i$-th component of the right hand side:
  - P -> DS { $$.list = $1.list; }
  - D -> var V; D { $$list = add_to_list($2.name, $4.list); }
  - S -> V := E; S { check($1.name, $$list); $$list = $$list; }
  - V -> x { $$name = "x"; }
  - y { $$name = "y"; }
  - z { $$name = "z"; }

Consider a LR parser ready to reduce using $\langle A \rangle ::= X_1 \ldots X_n$
- The symbols $X_i$ are on the stack before the reduction
- Previous reductions have associated semantic values (attributes) to these symbols
- They are then popped and $\langle A \rangle$ is pushed in their place
- While we do this, we execute some code that compute the attribute valued for $\langle A \rangle$
- In effect we have a syntactic stack (for the actual parsing) and a semantic stack (for the semantic values)
Verify that the grammar still generate the same language

**ISSUES IN BOTTOM-UP SYNTAX DIRECTED TRANSLATION**

- We require that the o-prefixed numbers be evaluated in octal
- Drawback: no restriction to octal digits for octal numbers
- Major drawback: not enough information from below for the differentiation between decimal and octal numbers
  - Semantic rules for computing these are different, yet they should all get attached to the rules for (int)
  - The decision on whether to process a decimal or octal number happens when o is shifted on the stack
  - At that time however an (int) has already been reduced and so its semantic actions have already been applied
  - In addition, semantic rules can only be applied to reductions, not shifts

**FIRST SOLUTION: RULE CLONING**

- Since our problem is caused by using the same rules for two different things, we can clone those rules so that we have separate copies for separate purposes
- When to use one set of rules and when to use the other is given based on the context of the nonterminal (i.e., where is the nonterminal used)

**SECOND SOLUTION: FORCING SEMANTIC ACTIONS**

- Suppose we need a semantic action when shifting some token x
  - We can insert a new rule (A) ::= x, and attach the action to this rule
  - All the occurrences of x in the original grammar will be replaced by (A)
- Suppose we need a semantic action between two symbols x and y
  - We then insert a new rule (A) ::= ϵ and attach the action to it
  - All the occurrences of x y in the original grammar will be replaced by x (A) y

**THIRD SOLUTION: GRAMMAR RESTRUCTURING**

- Global variables are undesirable because rules may be recursive and this may have unexpected consequences on these variables
  - Global variables can also make the semantic actions difficult to write and maintain since there is a lack of separation between actions
  - Proper initialization and resetting may be problematic
- A more robust solution is to restructure the parse tree as to eliminate the need for global variables:
  - Sketch a parse tree that allows bottom-up synthesis without global variables
  - Revise the grammar to achieve that parse tree
  - Verify that the grammar is still suitable for parsing (LALR(1), etc.)
  - Verify that the grammar still generate the same language

**Note the use of the global variable base (common occurrence)**

- The same caveats about modifying the grammar (semantic-only rules, equivalence) apply

\[
\begin{align*}
\langle \text{digit} \rangle &::= 0 \mid 1 \mid \ldots \mid 9 \\
\langle \text{int} \rangle &::= \langle \text{digit} \rangle \mid \langle \text{int} \rangle \langle \text{digit} \rangle \\
\langle \text{num} \rangle &::= o \langle \text{int} \rangle \mid \langle \text{int} \rangle
\end{align*}
\]
Top-down parsers are usually recursive descent parsers

- The computation of attributes is naturally inserted in the code, just like the code for constructing the AST
- Same ideas as above may be required to modify the grammar so that all the attributes can be computed

```cpp
class Node {
    ...
};

Node* Sequence() {
    Node* current = new Node(SEQ, ...);
    if (t == CLS_BRACE) /* <empty> */ ;
    else {
        /* <statement> <sequence> */
        current.addChild(Statement());
        current.addChild(Sequence());
    }
    return current;
}
```

Also see the example in the textbook

Abstract Syntax Trees

- The most common semantic actions are the ones that construct the abstract syntax tree for the input program
  - AST is a simplified and more compact representation of the parse tree
  - Just like in a parse tree, an AST node can have an arbitrary number of children
  - Links to the parent often needed (depending on the algorithms used in the semantic analysis)

- The data structure for an AST node can be approached in two ways
  - Have individual types for individual nodes (assignment, conditional, loop, etc.) → see assignments
    - Handy for languages that provide type definitions with inheritance, case in which this is the preferred method
    - Awkward in languages that do not offer inheritance constructs
  - Have the same data structure for all nodes
    - General, language-independent solution
    - Needs efficient representation for nodes with arbitrary number of children
    - Typical implementation: left-child-right-sibling
      - Each node is a node in a binary tree
      - The "left child" of a node points to the first child of that node
      - The "right child" of a node points to the next (right) sibling of that node

AST Design Principles

- AST design is crucial for the next phases of the compilation process
  - It should be possible to reconstitute ("unparse") the program from an AST
    - An AST node must hold enough information to recall the program fragment that generated it
  - Subsequent phases of the compilation process must access the AST through suitable interfaces
    - Different phases have different requirements (and so will use different interfaces)
    - Several phases will modify AST nodes
    - It is crucial to provide proper encapsulation to ensure that the AST information is not altered inadvertently
  - Subsequent compilation phases will traverse the AST (possibly repeatedly)
    - The easiest way to accomplish this is through polymorphic and recursive functions defined within the class hierarchy of AST node
      - The functions must be virtual to ensure the proper application for each node type
    - Most useful pattern for such functions: visitors → traverse the whole tree recursively