Attribute Grammars

What is an attribute grammar?

- A context-free grammar augmented with a set of semantic rules
- Each symbol in the derivation has a set of values, or attributes
- The semantic rules specify how to compute a value for each attribute

Example grammar:

\[
\begin{align*}
S & \rightarrow E \\
E & \rightarrow E + T \\
| & \quad E - T \\
| & \quad T \\
T & \rightarrow T * F \\
| & \quad T / F \\
| & \quad F \\
F & \rightarrow \text{num}
\end{align*}
\]

We want to write an expression interpreter

One way to do this is to augment the expression grammar with semantic rules that compute the value of each valid expression.
Expression Interpreter

### Productions

<table>
<thead>
<tr>
<th>S</th>
<th>→</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>E₀</td>
<td>→</td>
<td>E₁ + T</td>
</tr>
<tr>
<td></td>
<td></td>
<td>E₁ - T</td>
</tr>
<tr>
<td></td>
<td></td>
<td>T</td>
</tr>
<tr>
<td>T₀</td>
<td>→</td>
<td>T₁ * F</td>
</tr>
<tr>
<td></td>
<td></td>
<td>T₁ / F</td>
</tr>
<tr>
<td></td>
<td></td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>→</td>
<td>num</td>
</tr>
</tbody>
</table>

#### Semantic Rules

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>S.val</td>
<td>← E.val</td>
</tr>
<tr>
<td>E₀.val</td>
<td>← E₁.val + T.val</td>
</tr>
<tr>
<td>E₁.val</td>
<td>← E₁.val - T.val</td>
</tr>
<tr>
<td>T₀.val</td>
<td>← T₀.val * F.val</td>
</tr>
<tr>
<td>T₁.val</td>
<td>← T₁.val / F.val</td>
</tr>
<tr>
<td>F.val</td>
<td>← num.val</td>
</tr>
</tbody>
</table>

Notice that:

- Semantic rules use context information
- In this attribute grammar, attributes of grammar symbols on the lhs are computed using the attributes of grammar symbols on the rhs (such attributes are called synthesized attributes)
- The token `num` has a value attribute returned by the lexer

---

To Evaluate an Expression

For "10 – 2 * 3"

Annotated parse tree
(values of the attributes are evaluated)
Attribute Grammars

- Attributes are associated with nodes in the parse tree (terminals and non-terminals)
- Productions are associated with semantic rules which define how to assign values to attributes
- An attribute is defined (computed) only once, using local information
- Identical terms in a production are labeled for uniqueness
  - \( E \rightarrow E + T \) becomes \( E_0 \rightarrow E_1 + T \)
- Rules and parse tree define an attribute dependence graph
  - Dependence graph must be non-circular, otherwise it has no meaning
This produces a high-level, functional specification (no side-effects)

Synthesized attribute
- Depends on values from children

Inherited attribute
- Depends on values from siblings and parent

\[ A \rightarrow X_1 X_2 \ldots X_n \]

- **Synthesized attributes**: An attribute of \( A \) computed using attributes of \( X_1, X_2, \ldots, X_n \)
  - Example: \( E_0 \rightarrow E_1 + T \) \( \{ E_0.val \leftarrow E_1.val + T.val \} \)
    
    \begin{align*}
    \text{production} & \quad \text{semantic rule}
    \end{align*}

- **Inherited attributes**: An attribute of a symbol on the rhs computed using attributes of \( A, X_1, X_2, \ldots, X_n \)
  - Example: \( Decl \rightarrow Type \ L \ ; \ \{ \ L.type \leftarrow Type.type \} \)
    
    \begin{align*}
    \text{production} & \quad \text{semantic rule}
    \end{align*}

  - Example: \( L_0 \rightarrow L_1 \ , \ id \) \( \{ \ L_1.type \leftarrow L_0.type \} \)
    
    \begin{align*}
    \text{production} & \quad \text{semantic rule}
    \end{align*}
Another Example Grammar

This grammar describes signed binary numbers.

We would like to augment it with rules that compute the decimal value of each valid input string.

Example Parse Trees

For "-1"

For "-101"
### Attribute Grammar

Add rules to compute the decimal value of a signed binary number.

#### Productions

<table>
<thead>
<tr>
<th>Attribute Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Number</strong> → <strong>Sign</strong> <strong>List</strong>&lt;br&gt;<strong>List.pos</strong> ← 0&lt;br&gt;If <strong>Sign.neg</strong>&lt;br&gt;then <strong>Number.val</strong> ← - <strong>List.val</strong>&lt;br&gt;else <strong>Number.val</strong> ← <strong>List.val</strong></td>
</tr>
<tr>
<td><strong>Sign</strong> → +&lt;br&gt;<strong>Sign.neg</strong> ← false</td>
</tr>
<tr>
<td><strong>Sign</strong> → -&lt;br&gt;<strong>Sign.neg</strong> ← true</td>
</tr>
<tr>
<td><strong>List</strong>&lt;sub&gt;0&lt;/sub&gt; → <strong>List</strong>&lt;sub&gt;1&lt;/sub&gt; <strong>Bit</strong>&lt;br&gt;<strong>List</strong>&lt;sub&gt;0&lt;/sub&gt;.pos ← <strong>List</strong>&lt;sub&gt;0&lt;/sub&gt;.pos + 1&lt;br&gt;<strong>List</strong>&lt;sub&gt;0&lt;/sub&gt;.val ← <strong>List</strong>&lt;sub&gt;0&lt;/sub&gt;.val + <strong>Bit</strong>.val</td>
</tr>
<tr>
<td><strong>List</strong>&lt;sub&gt;0&lt;/sub&gt; → <strong>Bit</strong>&lt;br&gt;<strong>Bit.pos</strong> ← <strong>List</strong>.pos&lt;br&gt;<strong>List</strong>.val ← <strong>Bit</strong>.val</td>
</tr>
<tr>
<td><strong>Bit</strong> → 0&lt;br&gt;<strong>Bit.val</strong> ← 0</td>
</tr>
<tr>
<td><strong>Bit</strong> → 1&lt;br&gt;<strong>Bit.val</strong> ← 2&lt;sup&gt;<strong>Bit.pos</strong>&lt;/sup&gt;</td>
</tr>
</tbody>
</table>

#### Attributes

- **Number.val**
- **Sign.neg**
- **List.pos**
- **List.val**
- **Bit.pos**
- **Bit.val**

### Example

#### For “-1”

Evaluation order for attributes:
- Independent attributes first
- Others in order as input values become available

One possible evaluation order:
1. **List.pos**
2. **Sign.neg**
3. **Bit.pos**
4. **Bit.val**
5. **List.val**
6. **Number.val**

Other orders are possible

Evaluation order must be consistent with the attribute dependence graph
Example: Dependency Graph For Attributes

This is the complete attribute dependence graph for "-101".
It shows the flow of all attribute values in the example.
Some flow downward (or sideways) → inherited attributes
Some flow upward → synthesized attributes
A rule may use attributes in the parent, children, or siblings of a node

Evaluation Methods

Dynamic, dependence-based methods
• Build the parse tree
• Build the dependence graph
• Topological sort the dependence graph
• Compute the attributes in topological order

Rule-based methods
• Analyze rules at compiler-generation time
• Determine a fixed (static) ordering
• Evaluate nodes in that order

Oblivious methods
• Ignore rules and parse tree
• Pick a convenient order (at design time) and use it
• This is what JavaCUP, Yacc and Bison do
For 

"-101"

If we show the computation …

and then peel away the parse tree …

All that is left is the attribute dependence graph.
This succinctly represents the flow of values in the problem instance.
The dynamic methods start with the independent values, and then follow the graph edges.
The rule-based methods try to discover "good" orders by analyzing the rules.
The oblivious methods ignore the structure of this graph.

The dependence graph must be acyclic
S-Attributed Grammars

- A grammar that uses only synthesized attributes is called an:
  - *S-attributed grammar*
  - S-attributed grammars can be evaluated in a single bottom-up pass

- LR parsers can easily deal with S-attributed grammars
  - Store the attributes of the symbols in the parser stack
  - When a reduce action is taken
    - Symbols in the rhs of the production and their attributes are already in the stack
    - Compute the synthesized attributes of the symbol in the lhs of the production using the attributes of the symbols on the rhs

### Synthesized Attributes on the Parser Stack

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E_0 \rightarrow E_1 + T$</td>
<td>$E_0.val \leftarrow E_1.val + T.val$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$T$</th>
<th>$T.val$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$+$</td>
<td></td>
</tr>
<tr>
<td>$E_1$</td>
<td>$E_1.val$</td>
</tr>
</tbody>
</table>

Top of the parser stack

After the reduction:

| $E_0$ | $E_0.val$ |

Top of the parser stack
S-Attributed Grammar

Evaluation Example

**Input:** <num,10> –<num,2> * <num,3>

**Evaluation of attributes during LR parsing:**

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>num – num * num $</td>
</tr>
<tr>
<td>$ (num,10)</td>
<td>– num * num $</td>
</tr>
<tr>
<td>$ (F,10)</td>
<td>– num * num $</td>
</tr>
<tr>
<td>$ (T,10)</td>
<td>– num * num $</td>
</tr>
<tr>
<td>$ (E,10)</td>
<td>– num * num $</td>
</tr>
<tr>
<td>$ (E,10) – (num,2)</td>
<td>* num $</td>
</tr>
<tr>
<td>$ (E,10) – (F,2)</td>
<td>* num $</td>
</tr>
<tr>
<td>$ (E,10) – (T,2)</td>
<td>* num $</td>
</tr>
<tr>
<td>$ (E,10) – (T,2) * num $</td>
<td></td>
</tr>
<tr>
<td>$ (E,10) – (T,2) * (num,3) $</td>
<td></td>
</tr>
<tr>
<td>$ (E,10) – (T,2) * (F,3) $</td>
<td></td>
</tr>
<tr>
<td>$ (E,10) – (T,6)</td>
<td>$</td>
</tr>
<tr>
<td>$ (E,10)</td>
<td>$</td>
</tr>
<tr>
<td>$ (S,10)</td>
<td>$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attribute grammar:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ S \rightarrow E $</td>
</tr>
<tr>
<td>$ E_0 \rightarrow E_1 + T $</td>
</tr>
<tr>
<td>$ E_1 \rightarrow -T $</td>
</tr>
<tr>
<td>$ T \rightarrow T_1 * F $</td>
</tr>
<tr>
<td>$ T_1 \rightarrow T_1 / F $</td>
</tr>
<tr>
<td>$ F \rightarrow num $</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Corresponding annotated parse tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ S \rightarrow E \rightarrow E_0 \rightarrow E_1 \rightarrow E_1 \rightarrow T \rightarrow E \rightarrow T_1 \rightarrow F \rightarrow F ]</td>
</tr>
</tbody>
</table>

L-Attributed Grammars

- If inherited attribute of a symbol is computed using the inherited attributes of its parent and attributes of symbols on its left in the production, then the grammar is called an: **L-attributed grammar**

- Given a symbol $X_i$ on the rhs of production $A \rightarrow X_1, X_2, ..., X_n$, each inherited attribute of $X_i$ depends only on:
  - Inherited attributes of $A$
  - The attributes of $X_1, X_2, ..., X_{i-1}$ to the left of $X_i$ on the rhs of the production
L-Attributed Grammars

- L-attributed grammars can be evaluated using a depth-first traversal of the parse tree:

```plaintext
procedure dfsvisit(n: node) begin
  for each child m of n from left to right do
    evaluate inherited attributes of m;
    dfsvisit(m);
  endfor
  evaluate synthesized attributes of n
end
```

The nodes in this algorithm are the nodes of the parse tree. Start the depth-first traversal by calling the `dfsvisit` on the root of the parse tree.

**L-Attributed Grammar Evaluation Example**

**Attribute grammar:**

```
Decl → Type L
L₀ → L₁, id | id
Type → boolean | integer
```

**Input:** 

"integer id, id, id"

**Annotated parse tree:**

```
<table>
<thead>
<tr>
<th>Node</th>
<th>label</th>
<th>type</th>
<th>type</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Decl</td>
<td>type</td>
<td>integer</td>
<td>integer</td>
</tr>
<tr>
<td>2</td>
<td>Type</td>
<td>type</td>
<td>boolean</td>
<td>boolean</td>
</tr>
<tr>
<td>3</td>
<td>L</td>
<td>type</td>
<td>integer</td>
<td>integer</td>
</tr>
<tr>
<td>4</td>
<td>integer</td>
<td>type</td>
<td>integer</td>
<td>integer</td>
</tr>
<tr>
<td>5</td>
<td>L</td>
<td>type</td>
<td>integer</td>
<td>integer</td>
</tr>
<tr>
<td>6</td>
<td>id</td>
<td>type</td>
<td>integer</td>
<td>integer</td>
</tr>
<tr>
<td>7</td>
<td>L</td>
<td>type</td>
<td>integer</td>
<td>integer</td>
</tr>
<tr>
<td>8</td>
<td>id</td>
<td>type</td>
<td>integer</td>
<td>integer</td>
</tr>
<tr>
<td>9</td>
<td>id</td>
<td>type</td>
<td>integer</td>
<td>integer</td>
</tr>
</tbody>
</table>
```

**Depth First Evaluation:**

```
dfsvisit(1)
dfsvisit(2)
dfsvisit(4)
end dfsvisit(4)
compute Type.type
dfsvisit(2)
calculate L.type
dfsvisit(3)
calculate L.type
dfsvisit(5)
calculate L.type
dfsvisit(7)
calculate id.type
dfsvisit(9)
calculate id.type
dfsvisit(8)
calculate id.type
dfsvisit(6)
dfsvisit(5)
calculate id.type
dfsvisit(3)
dfsvisit(1)
```
An Extended Attribute-Grammar Example

Grammar for a block of assignments

```
Block_0 → Block_1 Assign
     | Assign
Assign → ident = Expr ;
Expr_0 → Expr_1 + Term
     | Expr_1 - Term
     | Term
Term_0 → Term_1 * Factor
     | Term_1 / Factor
     | Factor
Factor → ( Expr )
     | Number
     | Identifier
```

Estimate execution time
* Each operation has a COST
* Add them, bottom up
* Assume a load per value
* Assume no reuse
Can be solved using an attribute grammar

An Extended Example (continued)

```
Block_0 → Block_1 Assign
     | Assign
Assign → ident = Expr ;
Expr_0 → Expr_1 + Term
     | Expr_1 - Term
     | Term
Term_0 → Term_1 * Factor
     | Term_1 / Factor
     | Factor
Factor → ( Expr )
     | Number
     | Identifier
```

Block_0.cost ← Block_1.cost + Assign.cost
Block_0.cost ← Assign.cost
Assign.cost ← COST(store) + Expr.cost
Expr_0.cost ← Expr_1.cost + COST(add) + Term.cost
Expr_0的成本 ← Expr_1的成本 + COST(个) + Term的成本
Term_0.cost ← Term_1.cost + COST(mult) + Factor.cost
Term_0的成本 ← Term_1的成本 + COST(乘) + Factor的成本
Factor.cost ← Factor.cost
Factor.cost ← COST(loadi)
Factor成本 ← Factor成本
Factor成本 ← COST(load)

All the attributes are synthesized!
An Extended Example

Properties of the example grammar
• All attributes are synthesized ⇒ S-attributed grammar
• Rules can be evaluated bottom-up in a single pass
  – Good fit to bottom-up, shift/reduce parser
• Easily understood solution
• Seems to fit the problem well

What about an improvement?
• Values are loaded only once per block (not at each use)
• Need to track which values have been already loaded

A Better Execution Model

Adding load tracking
• Need sets Before and After for each production
• Must be initialized, updated, and passed around the tree

```
<table>
<thead>
<tr>
<th>Factor \rightarrow ( Expr )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Factor.cost \leftarrow Expr.cost ;</td>
</tr>
<tr>
<td>Expr.Before \leftarrow Factor.Before ;</td>
</tr>
<tr>
<td>Factor.After \leftarrow Expr.After</td>
</tr>
<tr>
<td>Factor.cost \leftarrow COST(load) ;</td>
</tr>
<tr>
<td>Factor.After \leftarrow Factor.Before</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Factor.cost \leftarrow COST(load) ;</td>
</tr>
<tr>
<td>Factor.After \leftarrow Factor.Before</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>If (Identifier.name \notin Factor.Before) then</td>
</tr>
<tr>
<td>Factor.cost \leftarrow COST(load) ;</td>
</tr>
<tr>
<td>Factor.After \leftarrow Factor.Before \cup Identifier.name</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>Factor.cost \leftarrow 0</td>
</tr>
<tr>
<td>Factor.After \leftarrow Factor.Before</td>
</tr>
</tbody>
</table>
```
A Better Execution Model

- Load tracking adds complexity
- But, most of it is in the “copy rules”
- Every production needs rules to copy Before and After

A sample production

```
Expr0 → Expr1 + Term

Expr0.cost ← Expr1.cost + COST(add) + Term.cost;
Expr0.BEFORE ← Expr1.BEFORE;
Term.BEFORE ← Expr1.AFTER;
Expr0.AFTER ← Term.AFTER
```

These copy rules multiply rapidly
Each creates an instance of the set
Lots of work, lots of space, lots of rules to write

An Even Better Model

What about accounting for finite register sets?
- Before and After must be of limited size
- Adds more complexity to Factor → Identifier

Jump from tracking loads to tracking registers is small
- Copy rules are already in place
- Some local code to perform the allocation

Tracking loads introduced Before and After sets and caused significant change in the attribute grammar
Attribute Grammars

- Non-local computation needed lots of supporting rules
- Complex local computation is relatively easy

The Problems
- Copy rules increase complexity
- Copy rules increase space requirements
  - Need copies of attributes
- After we write the attribute grammar, to evaluate the attributes
  - Must build the parse tree
  - Must traverse tree to evaluate the attributes

Addressing the Problem

Use rules with side effects, store the results in global variables
- This is called ad-hoc syntax directed translation
  - This is the approach we use when we write semantic actions in yacc
- For the previous example, use a table of names (symbol table)
  - Field in table for loaded/not loaded state
- Avoids all the copy rules, allocation and storage headaches
- All inter-assignment attribute flow is through table
  - Clean, efficient implementation
  - Good techniques for implementing the table
  - When its done, information is in the table!
  - Cures most of the problems
- This design violates the functional paradigm
Reworking the Example (with load tracking)
Using Ad-Hoc Syntax Directed Translation

<table>
<thead>
<tr>
<th>Rule</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block₀ → Block₁ Assign Assign</td>
<td>cost ← 0;</td>
</tr>
<tr>
<td>Assign</td>
<td>Assign Ident = Expr ;</td>
</tr>
<tr>
<td>Expr₀</td>
<td>Expr₁ + Term Expr₁ − Term Term</td>
</tr>
<tr>
<td>Term₀</td>
<td>Term₁ * Factor Term₁ / Factor Factor</td>
</tr>
<tr>
<td>Factor</td>
<td>( Expr ) Number Identifier</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>i ← hash(Identifier);</td>
</tr>
<tr>
<td></td>
<td>if (Table[i].loaded = false) then</td>
</tr>
<tr>
<td></td>
<td>cost ← cost + COST(load);</td>
</tr>
<tr>
<td></td>
<td>Table[i].loaded ← true;</td>
</tr>
</tbody>
</table>

Reality

Most parsers are based on this ad-hoc style of context-sensitive analysis using ad-hoc syntax directed translation

Advantages
- Addresses the shortcomings of the attribute grammars
- Efficient, flexible

Disadvantages
- Must write the code with little assistance
- Programmer deals directly with the details
Is It Really “Ad-hoc”? 

Relationship between what is used in practice (ad-hoc syntax directed translation) and attribute grammars

Similarities
• Both associate rules with productions
• Application order determined by tools, not author
• Abstract names for symbols

Differences
• Actions in ad-hoc method are applied as a unit; not true for attribute grammar rules
• Anything goes in ad-hoc actions; attribute grammar rules are functional

Translation Schemes

• Translation scheme is another name of ad-hoc syntax directed translation
• In attribute grammars the order of evaluation for the semantic rules are not shown
  – They are always listed to the right of the production
  – As we discussed earlier, in the most general case, we have to generate a dependency graph to figure out the order of evaluation
• In a translation scheme the order of evaluation for the semantic rules can be explicitly given by inserting them to the rhs of the productions with the following constraints:
  – An inherited attribute for a symbol on the rhs must be computed in an action before that symbol
  – An action must not refer to the synthesized attribute of a symbol to the right of the action
  – A synthesized attribute for the nonterminal on the left can only be computed after all attributes it references have been computed (we can always place it at the end of the rhs)
More On Translation Schemes

• If an attribute grammar is L-attributed, we can easily convert it to a translation scheme
  – Insert each semantic rule that computes an inherited attribute for a grammar symbol just to the left of that symbol in the right hand side
  – Insert the semantic rules that compute synthesized attributes to the very end of the right hand side

• Note that, for L-attributed grammars, above translation will result in a translation scheme that satisfies the constraint listed in the previous slide

Translation Schemes: An Example

<table>
<thead>
<tr>
<th>Attribute Grammar</th>
<th>Productions</th>
<th>Semantic rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Does not specify the evaluation order for the semantic rules</td>
<td>( D \rightarrow T ; L )</td>
<td>( L.in \leftarrow T.type )</td>
</tr>
<tr>
<td>• In the most general case, you need to construct the dependency graph to figure out the evaluation order</td>
<td>( T \rightarrow \text{int} ) ;</td>
<td>( T.type \leftarrow \text{integer} )</td>
</tr>
<tr>
<td></td>
<td>; float</td>
<td>( T.type \leftarrow \text{float} )</td>
</tr>
<tr>
<td></td>
<td>( L \rightarrow L_1, id ) ;</td>
<td>( L.in \leftarrow L.in ; \text{id.type} \leftarrow L.in )</td>
</tr>
<tr>
<td></td>
<td>; id</td>
<td>( \text{id.type} \leftarrow L.in )</td>
</tr>
</tbody>
</table>

Translation Scheme
• Semantic actions are inserted to the right-hand-sides of the productions
• Semantic actions are enclosed in \{ \} to distinguish them from the rest of the symbols
• The location of a semantic action indicates when it should be evaluated

\[
\begin{align*}
D & \rightarrow T \; \{ \; L.in \leftarrow T.type \; \} \; L \\
T & \rightarrow \text{int} \; \{ \; T.type \leftarrow \text{integer} \; \} \\
& \; | \; \text{float} \; \{ \; T.type \leftarrow \text{float} \; \} \\
L & \rightarrow \{ \; L.in \leftarrow L.in \; \} \; \{ \; L_i \; \text{id} \; \{ \; \text{id.type} \leftarrow L.in \; \} \\
& \; | \; \text{id} \; \{ \; \text{id.type} \leftarrow L.in \; \}
\end{align*}
\]
**Predictive Top-down Translators**

- Given a translation scheme with an underlying LL(1) grammar
  - We can construct a top-down recursive-descent parser which executes the semantic actions while it is parsing
  - The semantic actions will be inserted in the code of the recursive-descent parser based on where the semantic action occurs on the right hand side of the production

- If the grammar is LL(1) and the semantic rules are L-attributed than this would always work

**Constructing Top-down Translators from Translation Schemes**

1. Construct a procedure for each nonterminal $A$ that takes inherited attributes of $A$ as input and returns its synthesized attributes as output
2. In the procedure for nonterminal $A$, decide which production to use based on the lookahead
3. Code fragment associated with each production does the following:
   1. For a terminal symbol $X$ with synthesized attribute $s$ save synthesized attribute in a variable $X_s$; match terminal $X$;
   2. For a nonterminal $B$ put an assignment statement $B_s = B(B_{i1}, B_{i2}, \ldots, B_{iN})$;
      where $B_s$ is the local variable for the synthesized attribute of nonterminal $B$ and $B_{i1}, B_{i2}, \ldots, B_{iN}$ are local variables for $N$ inherited attributes of nonterminal $B$ (N could be 0 in which case procedure for $B$ has no input parameters)
   3. Insert the code for semantic actions into appropriate places in the parser (remember translation schemes show when the semantic actions should be executed)
Top-Down Translator Example

Translation Scheme:

\[
E \rightarrow T \{ R.i \leftarrow T.val \} R \{ E.val \leftarrow R.s \} \\
R \rightarrow + \{ R.i \leftarrow R.i + T.val \} R \{ R.s \leftarrow R.s \} \\
| - \{ R.i \leftarrow R.i - T.val \} R \{ R.s \leftarrow R.s \} \\
| \epsilon \{ R.s \leftarrow R.i \} \\
T \rightarrow (E) \{ T.val \leftarrow E.val \} \\
| \text{num} \{ T.val \leftarrow \text{num}.val \} \\
\]

A procedure of the top-down translator:

```c
int R(Ri) { // procedure for nonterminal R
    int R1i, Tval, Rs, Rs;
    switch (lookahead) {
        case(PLUS): match(PLUS); Tval = T(); Ri = Ri + Tval;
                    R1s = R(R1i); Rs = R1s; break;
        case(MINUS): match(MINUS); Tval = T(); Ri = Ri - Tval;
                    R1s = R(R1i); Rs = R1s; break;
        default: Rs = Ri;
    }
    return Rs;
}
```

```c
int E() { // procedure for nonterminal E
    int Ri, Tval, Eval, Rs;
    Tval = T(); Ri = Tval; Rs = R(Ri); Eval = Rs;
    return Eval;
}
```

```c
int T() { // procedure for nonterminal T
    int Tval, Eval, numval;
    switch (lookahead) {
        case(LPAREN): match(LPAREN); Eval = E();
                      match(RPAREN); Tval = Eval; break;
        case(NUM): numval = lookahead.value;
                   match(NUM); Tval = numval; break;
        default: error();
    }
    return Tval;
}
```