A Step-Back
Chapter 3

- Strings
- Regular expressions
- Tokens
- Transition diagrams
- Finite Automata
A Step-Back

Chapter 4

- Grammars
- Derivations
- Parse-trees
- Top-down parsing (LL)
- Bottom-up parsing (LR, SLR, LALR)
We Need Some Tools

• To help in semantic analysis
• To help in intermediate code generation
• Two such tools
  – Semantic rules (Syntax-Directed Definitions)

\[
\begin{align*}
  \text{Production} & \quad \text{Semantic Rule} \\
  E \to E_1 + T & \quad E.code = E_1.code \ | T.code \ | '+'
\end{align*}
\]

– Semantic actions (Syntax Directed Translations)

\[
E \to E_1 + T \quad \{ \text{print } '+' \} 
\]
Syntax-Directed Definitions

- Context-free grammar
- With attributes and rules to calculate the attributes

**Production**

\[ E \rightarrow E_1 + T \]

**Semantic Rule**

\[ E.code = E_1.code \| T.code \| '+' \]
Two Types of Attributes
Two Types of Attributes

Synthesized Attributes

Attribute of the node is defined in terms of:
• Attribute values at children of the node
• Attribute value at node itself

SDD involving only synthesized attributes is called $S$-attributed
Two Types of Attributes

Inherited Attributes

Attribute of the node is defined in terms of:
• Attribute values at parent of the node
• Attribute values at siblings
• Attribute value at node itself
A parse tree showing the values of its attributes is called **annotated parse tree**.
Example

Give the annotated parse tree of \((3+4)*(5+6)n\)

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) ( L \rightarrow E \ n )</td>
<td>( L.val = E.val )</td>
</tr>
<tr>
<td>2) ( E \rightarrow E_1 + T )</td>
<td>( E.val = E_1.val + T.val )</td>
</tr>
<tr>
<td>3) ( E \rightarrow T )</td>
<td>( E.val = T.val )</td>
</tr>
<tr>
<td>4) ( T \rightarrow T_1 * F )</td>
<td>( T.val = T_1.val \times F.val )</td>
</tr>
<tr>
<td>5) ( T \rightarrow F )</td>
<td>( T.val = F.val )</td>
</tr>
<tr>
<td>6) ( F \rightarrow ( \ E \ ) )</td>
<td>( F.val = E.val )</td>
</tr>
<tr>
<td>7) ( F \rightarrow \text{digit} )</td>
<td>( F.val = \text{digit.lexval} )</td>
</tr>
<tr>
<td>PRODUCTION</td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td></td>
</tr>
<tr>
<td>1) ( T \rightarrow FT' )</td>
<td></td>
</tr>
<tr>
<td>2) ( T' \rightarrow *FT_1' )</td>
<td></td>
</tr>
<tr>
<td>3) ( T' \rightarrow \epsilon )</td>
<td></td>
</tr>
<tr>
<td>4) ( F \rightarrow \text{digit} )</td>
<td></td>
</tr>
</tbody>
</table>
**Example**

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) $D \to TL$</td>
<td>$L.inh = T.type$</td>
</tr>
<tr>
<td>2) $T \to\text{int}$</td>
<td>$T.type = \text{integer}$</td>
</tr>
<tr>
<td>3) $T \to\text{float}$</td>
<td>$T.type = \text{float}$</td>
</tr>
</tbody>
</table>
| 4) $L \to L_1, \text{id}$ | $L_{1.inh} = L.inh$
   \quad addType(\text{id.entry}, L.inh)$ |
| 5) $L \to \text{id}$ | $addType(\text{id.entry}, L.inh)$ |

Give annotated parse-trees for:

```
int a, b, c
```
Evaluation Orders of SDDs

• Annotated parse tree shows attribute values
• **Dependency graph** helps us determine how those values are computed
Topological Order
Cycles

- Arbitrary SDDs can have cycles.
- Cycles need to be avoided
  - Cannot proceed
  - Detecting cycles has exponential time-complexity.
- Two type of SDDs guarantee no-cycles:
  - S-attributed
  - L-attributed
S-Attributed Definitions

• Every attribute is synthesized
• We can evaluate its attribute in any bottom-up order of the nodes of the parse tree
  (e.g. postorder traversal -> LR parser).
L-Attributed Definitions

• Dependency graph edges can only go from left to right
  – i.e. use attributes from above or from the left
### Example

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) $D \rightarrow TL$</td>
<td>$L.inh = T.type$</td>
</tr>
<tr>
<td>2) $T \rightarrow \text{int}$</td>
<td>$T.type = \text{integer}$</td>
</tr>
<tr>
<td>3) $T \rightarrow \text{float}$</td>
<td>$T.type = \text{float}$</td>
</tr>
</tbody>
</table>
| 4) $L \rightarrow L_1, \text{id}$ | $L_1.inh = L.inh$
  addType(id.entry, L.inh) |
| 5) $L \rightarrow \text{id}$ | $\text{addType}(\text{id}.entry, L.inh)$ |

![Diagram](image)
Syntax-Directed Translations

- Context-free grammar
- Can implement SDDs and also complement them
- Program fragments embedded within production bodies
  - called semantic rules
  - Can appear anywhere within the production body
- Typically implemented during parsing
- Steps are usually as follows
  - Build parse tree
  - perform actions as you traverse left-to-right, depth-first (preorder)
Implementing LR-Parsable and SDD is S-Attributed

• By far the simplest
  – Action is placed at the end of the production rule
  – Action executed with the reduction from body to head of the rule
Example

Given LR parsing that we have seen, where do we store the attributes?

\[
\begin{align*}
L & \rightarrow E \, n \quad \{ \text{print}(E.val); \} \\
E & \rightarrow E_1 + T \quad \{ E.val = E_1.val + T.val; \} \\
E & \rightarrow T \quad \{ E.val = T.val; \} \\
T & \rightarrow T_1 * F \quad \{ T.val = T_1.val \times F.val; \} \\
T & \rightarrow F \quad \{ T.val = F.val; \} \\
F & \rightarrow (E) \quad \{ F.val = E.val; \} \\
F & \rightarrow \text{digit} \quad \{ F.val = \text{digit.}lexval; \}
\end{align*}
\]
SDT plays well with Shift-Reduce LR parsing!

Stack

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>X.x</td>
<td>Y.y</td>
<td>Z.z</td>
</tr>
</tbody>
</table>

State/grammar symbol
Synthesized attribute(s)

**Production**

**Actions**

$L \rightarrow E \ n$

\{ print(stack[top - 1].val);
    top = top - 1; \}

$E \rightarrow E_1 + T$

\{ stack[top - 2].val = stack[top - 2].val + stack[top].val;
    top = top - 2; \}

$E \rightarrow T$

$T \rightarrow T_1 * F$

\{ stack[top - 2].val = stack[top - 2].val \times stack[top].val;
    top = top - 2; \}

$T \rightarrow F$

$F \rightarrow ( E )$

\{ stack[top - 2].val = stack[top - 1].val;
    top = top - 2; \}

$F \rightarrow \text{digit}$
Actions Inside Production

- Actions can be placed anywhere in the production rule.
- It is performed immediately after all symbols to its left are processed.
- Example: \( B \rightarrow X \{a\} Y \)

- If the parse is bottom-up, then we perform action \( a \) as soon as this occurrence of \( X \) appears on the top of the parsing stack.
- If the parse is top-down, we perform \( a \) just before we attempt to expand this occurrence of \( Y \) (if \( Y \) a nonterminal) or check for \( Y \) on the input (if \( Y \) is a terminal).
Implementing L-Attributed SDDs

• L-attributed definitions can be used in many translation applications

• Several methods of implementation
  – Build parse tree and annotate
  – Build parse tree, add actions, execute in preorder
  – Recursive descent
Recursive Descent

• Function A for each nonterminal A
• Arguments of A are inherited attributes of nonterminal A
• Return value of A is the collection of synthesized attributes of A
Example

\[ S \rightarrow \textbf{while} \ (C) \ S_1 \quad \text{For that rule we want to generate labels:} \]
\[ \quad \text{L1: C} \]
\[ \quad \text{L2: S1} \]

\[ S \rightarrow \textbf{while} \ (C) \ S_1 \quad L1 = \textit{new}(); \]
\[ \quad L2 = \textit{new}(); \]
\[ \quad S_1.\text{next} = L1; \]
\[ \quad C.\textit{false} = S.\text{next}; \]
\[ \quad C.\textit{true} = L2; \]
\[ \quad S.\text{code} = \textit{label} \ || \ L1 \ || \ C.\text{code} \ || \ \textit{label} \ || \ L2 \ || \ S_1.\text{code} \]
Example

For that rule we want to generate labels:
L1: C
L2: S1

```c
string S(label next) {
    string Scode, Ccode; /* local variables holding code fragments */
    label L1, L2; /* the local labels */
    if ( current input == token while ) {
        advance input;
        check '(' is next on the input, and advance;
        L1 = new();
        L2 = new();
        Ccode = C(next, L2);
        check ')' is next on the input, and advance;
        Scode = S(L1);
        return("label" || L1 || Ccode || "label" || L2 || Scode);
    }
    else /* other statement types */
}
```
Example

\[ S \rightarrow \textbf{while} \ (C) \ S_1 \]

For that rule we want to generate labels:

L1: C
L2: S1

\begin{verbatim}
string S(label next) {
    string Scode, Ccode; /* local variables holding code fragment */
    label L1, L2; /* the local labels */
    if ( current input == token while ) {
        advance input;
        check '(' is next on the input, and advance;
        L1 = new();
        L2 = new();
        Ccode = C(next, L2);
        check ')' is next on the input, and advance;
        Scode = S(L1);
        return("label" || L1 || Ccode || "label" || L2 || Scode)
    }
    else /* other statement types */
}

void S(label next) {
    label L1, L2; /* the local labels */
    if ( current input == token while ) {
        advance input;
        check '(' is next on the input, and advance;
        L1 = new();
        L2 = new();
        print("label", L1);
        C(next, L2);
        check ')' is next on the input, and advance;
        print("label", L2);
        S(L1);
    }
    else /* other statement types */
}
\end{verbatim}
Reading

• SDD and SDT are complementary
• SDT can be used with two SDD trees
  – S-attribute
  – L-attribute
• Skim: 5.3, 5.4.3, 5.4.4, 5.4.5, 5.5.3, and 5.5.4
• Read the rest