Attributes and Definitions

- **S-Attributed Definitions**: Where all attributes are synthesized.
- **L-Attributed Definitions**: Where all inherited attributes are such that their values depend only on inherited attributes of the parent, and attributes of left siblings.

---

Synthesized Attributes and Bottom-up Parsing

Keep track of attributes of symbols while parsing.

- **Shift-reduce parsing**: Keep a stack of attributes corresponding to stack of symbols. Compute attributes of LHS symbol while performing reduction (i.e., while pushing the symbol on symbol stack).

---

Syntax Directed Definition

```plaintext
E  \rightarrow E \cdot E
E  \rightarrow E + E
E  \rightarrow \text{int}

E  \rightarrow E_1 \cdot E_2 \quad \{E,\text{val} = E_1,\text{val} \cdot E_2,\text{val}\}
E  \rightarrow E_1 + E_2 \quad \{E,\text{val} = E_1,\text{val} + E_2,\text{val}\}
E  \rightarrow \text{int} \quad \{E,\text{val} = \text{int},\text{val}\}
```

---

Attributes

- **Synthesized**: Attribute of LHS symbol of a grammar rule, whose value depends on attributes of RHS symbols of the grammar rule. Value flows from child to parent in the parse tree.
- **Inherited**: Attribute of an RHS symbol of a grammar rule, whose value depends on attributes of the LHS symbol and the other RHS symbols of the grammar rule. Value flows into a node in the parse tree from parents and/or siblings.
Top-down Parsing

```c
parse_S() {
    /* production 1 */
    parse_S();
    parse_S();
    return;
}
```

```c
parse_B() {
    consume(OPN_BRACE);
    parse_S();
    consume(CLOSE_BRACE);
}
```

---

Inherited Attributes and Top-down Parsing

```c
parse_S(block) {
    /* production 1 */
    parse_S(block);
    parse_S(block);
    return;
}
```

```c
parse_B(block) {
    consume(OPN_BRACE);
    parse_S(child[block]);
    consume(CLOSE_BRACE);
}
```

---

Synthesized Attributes & Shift-reduce parsing

```
E → E+E
E → E*E
E → int
```

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input Stream</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>3 * 2 + 5 $</td>
<td>$</td>
</tr>
<tr>
<td>$</td>
<td>int</td>
<td>* 2 + 5 $</td>
</tr>
<tr>
<td>$ E</td>
<td>* 2 + 5</td>
<td></td>
</tr>
<tr>
<td>$ E *</td>
<td>2 + 5</td>
<td></td>
</tr>
<tr>
<td>$ E * int</td>
<td>+ 5 $ $ 3 + 2</td>
<td></td>
</tr>
<tr>
<td>$ E * E</td>
<td>+ 5 $ $ 3 + 2</td>
<td></td>
</tr>
<tr>
<td>$ E</td>
<td>* 5 $</td>
<td></td>
</tr>
<tr>
<td>$ E +</td>
<td>5 $ $ 6 + 5</td>
<td></td>
</tr>
<tr>
<td>$ E + int</td>
<td>$ 6 + 5</td>
<td></td>
</tr>
<tr>
<td>$ E + E</td>
<td>$ $ 6 + 5</td>
<td></td>
</tr>
<tr>
<td>$ E</td>
<td>$ $ 11</td>
<td></td>
</tr>
</tbody>
</table>

---

Inherited Attributes

```
Ss → S; Ss
Ss → ε
B → { Ss }
S → B
S → other
```

- $Ss → S; Ss$
  - $Ss, block = Ss.block$
  - $Ss_1, block = Ss.block$
- $Ss → ε$
- $B → \{ Ss \}$
  - $Ss, block = child(B.block)$
- $S → B$
  - $B, block = S.block$
- $S → other$
  - $other, block = S.block$
Inherited Attributes and Bottom-up Parsing

Inherited attributes depend on the context in which a symbol is used.

For inherited attributes, we cannot assign an value to a node's attributes unless the parent's attributes are known.

When building parse trees bottom-up, parent of a node is not known when the node is created!

Solution: Ensure that all attributes are inherited only from left siblings,

Use "global" variables to capture inherited values,

and introduce "marker" nonterminals to manipulate the global variables.

---

Attributes and Top-down Parsing

- Inherited: analogous to function arguments
- Synthesized: analogous to return values

L-attribute definitions mean that argument to a parsing function is

- argument of the calling function, or
- return value/argument of a previously called function

---

Inherited Attributes & Bottom-up parsing

\[
\begin{align*}
S_s & \rightarrow S; S_s \\
S_s & \rightarrow \epsilon \\
B & \rightarrow \{ S_s \} \\
S & \rightarrow B \\
S & \rightarrow \text{other}
\end{align*}
\]

\[
\begin{align*}
B & \rightarrow \{ M_1 S_s M_2 \} \\
M_1 & \rightarrow \epsilon \quad \{ \text{current block + ;} \} \\
M_2 & \rightarrow \epsilon \quad \{ \text{current block - ;} \}
\end{align*}
\]
Construction of Abstract Syntax Trees

Typically done simultaneously with parsing
... as another instance of syntax-directed translation
... for translating concrete syntax (the parse tree) to abstract syntax (AST).
... with AST as a synthesized attribute of each grammar symbol.

Semantic Analysis Phases of Compilation

- Build an Abstract Syntax Tree (AST) while parsing
- Decorate the AST with type information (type checking/inference)
- Generate intermediate code from AST
  - Optimize intermediate code
  - Generate final code

Abstract Syntax Tree (AST)

Represents the syntactic structure of the input program, independent of peculiarities in the grammar.

An Example:

Consider a statement of the form
"if (m == 0) S1 else S2"
where S1 and S2 stand for some block of statements.
A possible AST for this statement is:

```
if-then-else
  m
  0
  AST for S1
  AST for S2
```