5.1 Syntax-Directed Definitions

- Each grammar symbol has two kinds of associated attributes:
  - *Synthesized*: available from children (RHS) of grammar rule; e.g., for $E \rightarrow E_1 + T$, have $E.val = E_1.val + T.val$
  - *Inherited*: available from siblings or parent of RHS symbols; e.g., for ML `decl` → `id` : `type`, `id` gets its type from `type`.

- Attributes of terminals are specified by the lexical analyzer.

Syntax-Directed Translation

- Touched on in section 2.3
- Useful for doing things after parsing (type checking, code generation)
- Basic idea: attach attributes to grammar symbols, then do something with attributes when they appear in parse tree.

5.1 Synthesized Attributes

- Evaluated bottom-up
- If only synthesized attributes are used, definition is called an *S-attributed definition*
- E.g.

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E \rightarrow E_1 + T$</td>
<td>$E.val := E_1.val + T.val$</td>
</tr>
<tr>
<td>$E \rightarrow T^*$</td>
<td>$E.val := T.val$</td>
</tr>
<tr>
<td>$T \rightarrow F^*$</td>
<td>$T.val := F.val$</td>
</tr>
<tr>
<td>$F \rightarrow (E)$</td>
<td>$F.val := E.val$</td>
</tr>
<tr>
<td>$F \rightarrow \text{digit}$</td>
<td>$F.val := \text{digit}.lexval$</td>
</tr>
</tbody>
</table>
Synthesized Attributes

• Evaluating expression $3 \times 5 + 4$

```
E
  /|
 / |
/  |
T  F
```

```
F
```

```
T
```

```
*  F
  |
  |
  F
```

```
digit.lexval = 4
```

```
digit.lexval = 3
```

```
F
```

```
digit.lexval = 5
```

```
F
```

```
digit.lexval = 5
```

```
digit.lexval = 3
```

```
digit.lexval = 3
```

```
E
  /|
 / |
/  |
T  F
```

```
T
```

```
*  F
  |
  |
  F
```

```
digit.lexval = 4
```

```
digit.lexval = 3
```

```
digit.lexval = 5
```

```
digit.lexval = 3
```

```
digit.lexval = 4
```

```
digit.lexval = 5
```

```
digit.lexval = 3
```

```
digit.lexval = 3
```
Synthesized Attributes

• Evaluating expression $3 \times 5 + 4$

```
   E
  / \
 T   +
   \|
   F

T.val = 3
F.val = 3
digit.lexval = 3
```

```
T.val = 3
F.val = 5
digit.lexval = 5
```

```
T.val = 3
digit.lexval = 3
```

```
E
/ \
T   +
   \|
   F

T.val = 3
F.val = 5
digit.lexval = 4
```

```
T.val = 3
F.val = 5
digit.lexval = 5
```

```
T.val = 3
digit.lexval = 3
```

```
E
/ \
T   +
   \|
   F

T.val = 4
F.val = 4
digit.lexval = 4
```

```
T.val = 15
F.val = 5
digit.lexval = 4
```

```
T.val = 4
digit.lexval = 4
```

```
E
/ \
T   +
   \|
   F

T.val = 15
F.val = 5
digit.lexval = 4
```

```
T.val = 15
F.val = 5
digit.lexval = 5
```

```
T.val = 4
digit.lexval = 4
```
5.2 Construction of Syntax Trees

- Parse trees may be too concrete – e.g., don’t need keywords if, then, else, knowing that you have a conditional
- Syntax Tree (a.k.a. Abstract Syntax Tree) encodes only essential information

```
      expr
    /   \
  if    stmt  else stmt
    \   /   /   /
  expr stmt stmt
```

PARSE TREE  SYNTAX TREE

### Synthesized Attributes

- Evaluating expression $3 \times 5 + 4$

```
E \cdot val = 19
\downarrow
E \cdot val = 15
\downarrow
T \cdot val = 15
\downarrow
F \cdot val = 3
\downarrow
digit.lexval = 3
```

```
T \cdot val = 15
\downarrow
F \cdot val = 4
\downarrow
digit.lexval = 4
```

```
F \cdot val = 3
\downarrow
digit.lexval = 5
```

### Inherited Attributes

- Evaluated top-down using inherited attribute in
- Useful for assigning attributes based on context; e.g., typed variable declarations in Java:

```
int a, b, c;
```

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D \rightarrow T \ L$</td>
<td>$L_{\text{in}} := T_{\text{type}}$</td>
</tr>
<tr>
<td>$T \rightarrow \text{int}$</td>
<td>$T_{\text{type}} := \text{integer}$</td>
</tr>
<tr>
<td>$T \rightarrow \text{double}$</td>
<td>$T_{\text{type}} := \text{double}$</td>
</tr>
<tr>
<td>$L \rightarrow L_1 \cdot \text{id}$</td>
<td>$L_{\text{in}} := L_{\text{in}} : \text{addtype (id.entry, Lin)}$</td>
</tr>
<tr>
<td>$L \rightarrow \text{id}$</td>
<td>$\text{addtype (id.entry, Lin)}$</td>
</tr>
</tbody>
</table>

where addtype is a symbol-table method

### Syntax Trees for Expressions

- For expressions, make operator leaf nodes of parse tree into interior nodes of AST:

```
\frac{E}{T} + \frac{F}{3} \rightarrow \frac{\ast}{4} 5
```

```
\frac{E}{T} + \frac{F}{3} \rightarrow \frac{\ast}{4} 5
```
5.3 Bottom-Up Evaluation of S-Attributed Definitions

- Don't have to construct explicit parse tree: LR parser can do semantic actions when a reduction is made

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>id</td>
<td>shift</td>
</tr>
<tr>
<td>0 id 5</td>
<td>+ id 5</td>
<td>reduce F.ptr := mkleaf(id, id.entry)</td>
</tr>
<tr>
<td>0 F 3</td>
<td>* id 5</td>
<td>reduce T.ptr := F.ptr</td>
</tr>
<tr>
<td>0 F 2</td>
<td>* id 5</td>
<td>shift</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0 E 1 + 6 T 9</td>
<td>$</td>
<td>reduce E.ptr := mknode('+', E, T, T.ptr)</td>
</tr>
<tr>
<td>0 E 1</td>
<td>$</td>
<td>accept</td>
</tr>
</tbody>
</table>

- These semantic actions go with productions in yacc/CUP

Replacing Inherited by Synthesized Attributes (end of 5.6)

- Inherited attributes (e.g., type declarations) cannot be built bottom-up, because they come from node's parent/siblings

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>D → L : T</td>
<td>Lin := T.type</td>
</tr>
<tr>
<td>L → id</td>
<td>addtype (id.entry, Lin)</td>
</tr>
<tr>
<td>T → int</td>
<td>T.type := integer id real</td>
</tr>
<tr>
<td>T → real</td>
<td>T.type := real</td>
</tr>
</tbody>
</table>

Replacing Inherited by Synthesized Attributes (end of 5.6)

- But we can sometimes re-write the grammar to make the sibling a descendant

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>D → id L</td>
<td>addtype (id.entry, Lin)</td>
</tr>
<tr>
<td>L → : T</td>
<td>Lin := T.type</td>
</tr>
<tr>
<td>T → int</td>
<td>T.type := integer</td>
</tr>
<tr>
<td>T → real</td>
<td>T.type := real</td>
</tr>
</tbody>
</table>

- In a compiler written in Java, mknode, mkleaf would be constructors

Syntax Trees for Expressions

- Grammar symbols get pointers to AST nodes, so nodes can be passed along as AST is built:

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>E → E₁ + T</td>
<td>E.ptr := mknode('+', E₁.ptr, T.ptr)</td>
</tr>
<tr>
<td>E → T</td>
<td>E.ptr := T.ptr</td>
</tr>
<tr>
<td>T → T₁ * F</td>
<td>T.ptr := mknode('*', T₁.ptr, F.ptr)</td>
</tr>
<tr>
<td>T → F</td>
<td>T.ptr := F.ptr</td>
</tr>
<tr>
<td>F → ( E )</td>
<td>F.ptr := E.ptr</td>
</tr>
<tr>
<td>F → num</td>
<td>F.ptr := mkleaf(num, num.val)</td>
</tr>
<tr>
<td>F → id</td>
<td>F.ptr := mkleaf(id, id.entry)</td>
</tr>
</tbody>
</table>

- In a compiler written in Java, mknode, mkleaf would be constructors