Phases of a compiler

Intermediate Representation (IR): specification and generation

Figure 1.6, page 5 of text
Intermediate Representations
Directed Acyclic Graph (DAG)

- Similar to a syntax tree
- No repeated nodes: structure sharing
Ex. 6.1 [p 359]

\[ a + a \times (b - c) + (b - c) \times d \]
Ex. 6.1 [p 359]

\[ a + a \cdot (b - c) + (b - c) \cdot d \]
Ex. 6.1 [p 359]

\[ a + a \times (b - c) + (b - c) \times d \]
Ex. 6.1 [p 359]

\[ a + a * (b - c) + (b - c) * d \]

Things can be more complicated if expressions have side effects
### SDT

**Tree or DAG**

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 ( E \rightarrow E_1 + T )</td>
<td>( E.node = \text{new Node('+'}, E.node, T.node) )</td>
</tr>
<tr>
<td>2 ( E \rightarrow E_1 - T )</td>
<td>( E.node = \text{new Node('-'}, E.node, T.node) )</td>
</tr>
<tr>
<td>3 ( E \rightarrow E_1 \ast T )</td>
<td>( E.node = \text{new Node('*'}, E.node, T.node) )</td>
</tr>
<tr>
<td>4 ( E \rightarrow T )</td>
<td>( E.node = T.node )</td>
</tr>
<tr>
<td>5 ( T \rightarrow (E) )</td>
<td>( T.node = E.node )</td>
</tr>
<tr>
<td>6 ( T \rightarrow \text{id} )</td>
<td>( T.node = \text{new Leaf(id, id.entry)} )</td>
</tr>
<tr>
<td>7 ( T \rightarrow \text{num} )</td>
<td>( T.node = \text{new Leaf(num, num.val)} )</td>
</tr>
</tbody>
</table>

*Figure 6.4 in text (p. 360), corrected according to errata sheet.*
SDT
Tree or DAG

- SDT produces a tree if each call to Node creates a new tree node.
- SDT produces a DAG if for each call to Node there is a check whether this node already exists, and if so it returns a reference to the existing node rather than returning a new node.
Example

\[
\begin{align*}
    p_1 &= \text{Leaf}(id, \text{entry-a}) \\
    p_2 &= \text{Leaf}(id, \text{entry-a}) = p_1 \\
    p_3 &= \text{Leaf}(id, \text{entry-b}) \\
    p_4 &= \text{Leaf}(id, \text{entry-c}) \\
    p_5 &= \text{Node}('-', p_3, p_4) \\
    p_6 &= \text{Node}('*', p_1, p_5) \\
    p_7 &= \text{Node}('+', p_1, p_6) \\
    p_8 &= \text{Leaf}(id, \text{entry-b}) = p_3 \\
    p_9 &= \text{Leaf}(id, \text{entry-c}) = p_4 \\
    p_{10} &= \text{Node}('-', p_3, p_4) = p_5 \\
    p_{11} &= \text{Leaf}(id, \text{entry-d}) \\
    p_{12} &= \text{Node}('*', p_5, p_{11}) \\
    p_{13} &= \text{Node}('+', p_7, p_{12})
\end{align*}
\]
Value-number method
Algorithm 6.3 [p. 361]

- **Input:** label op, node l, node r
- **Output:** The value number of a node in the array with signature <op,l,r>
- **Method:** Search the array for a node M with signature <op,l,r>. If there is such a node, return the value number of M. If not, create in the array a new node N with signature <op,l,r> and return its value number.
Revisiting 6.1
see construction steps in figure 6.5 [p. 360]

<table>
<thead>
<tr>
<th></th>
<th>id</th>
<th>→ to ST entry for</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>id</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>id</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>id</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>id</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>
Three address code representation

\[ t_1 = b - c \]
\[ t_2 = a \times t_1 \]
\[ t_3 = a + t_2 \]
\[ t_4 = t_1 \times d \]
\[ t_5 = t_3 - t_4 \]
Three address code instructions
(see 6.2.1, pages 364-5)

1. \( x = y \text{ op } z \)
2. \( x = \text{ op } y \)   (treat \( \text{i2r} \) and \( \text{r2i} \) as unary ops)
3. \( x = y \)
4. \( \text{goto } L \)
5. \( \text{if } x \text{ goto } L \) / \( \text{ifFalse } x \text{ goto } L \)
6. \( \text{if } x \text{ relop } y \text{ goto } L \)
7. function calls:
   - \( \text{param } x \)
   - \( \text{call } p, n \)
   - \( y = \text{call } p \)
   - \( \text{return } y \)
8. \( x = y[i] \) and \( x[i] = y \)
9. \( x = &y, x = *y, *x = y \)