Examples Discussed in Class

CSE 6341

Operator Precedence and Associativity

We discussed the following ambiguous context-free grammar:

\[
\begin{align*}
\langle expr \rangle &::= \langle term \rangle \mid \langle expr \rangle + \langle expr \rangle \mid \langle expr \rangle * \langle expr \rangle \\
\langle term \rangle &::= x \mid y \mid z \mid (\langle expr \rangle)
\end{align*}
\]

As an example, for \( x+y+z \) this grammar allows two different parse trees. In one tree the first + is “deeper” in the tree (i.e., the + operator is left-associative). The resulting assembly code produced by a compiler may look something like

```
ADD R1, x, y
ADD R2, R1, z
```

Here \( R1 \) and \( R2 \) are registers. The first instruction adds the values in variables \( x \) and \( y \) and stores the result in \( R1 \). The second instruction adds the values in \( R1 \) and \( z \) and stores the result in \( R2 \). In the other possible parse tree, the second + is deeper (i.e., the + operator is right-associative). The assembly code may look something like

```
ADD R1, y, z
ADD R2, x, R1
```

As another example, for \( x+y*z \) this grammar allows two parse tree. For the tree where the + is deeper, the + operator has higher precedence than the * operator. For the tree where the * is deeper, the * operator has higher precedence than the + operator. Clearly, these different parse trees result in different computed values: \( (v_1 + v_2)v_3 \) vs \( v_1 + v_2v_3 \), where \( v_1 \) denotes the value stored in variable \( x \), etc.

In real languages, operator precedence and associativity is well-defined: for example, + and * are left-associative, and * has higher precedence. To remove the ambiguity and to achieve this precedence and associativity, the grammar can be modified as follows:

\[
\begin{align*}
\langle expr \rangle &::= \langle term \rangle \mid \langle expr \rangle + \langle term \rangle \\
\langle term \rangle &::= \langle factor \rangle \mid \langle term \rangle * \langle factor \rangle \\
\langle factor \rangle &::= x \mid y \mid z \mid (\langle expr \rangle)
\end{align*}
\]

Simple Expression Language with \texttt{let}

Consider a slightly generalized version of the attribute grammar for simple expressions, based on the following context-free grammar:
\( (S) ::= \langle E \rangle \)
\( (E)_1 ::= \text{const} \mid \langle I \rangle \mid (\langle E \rangle_2 + \langle E \rangle_3) \mid \text{let } \langle I \rangle = \langle E \rangle_2 \text{ in } \langle E \rangle_3 \text{ end} \)
\( (I) ::= \text{id} \)

Assume that terminal \text{const} represents integer constants and has an attribute \text{lexval} of type integer, representing the value of the constant. Similarly to \text{id.lexval}, the value of \text{const.lexval} is initialized by the lexical analyzer (also referred to as “scanner”). The evaluation rule for \( (E)_1 ::= \text{const} \) is as expected: \( (E)_1.val := \text{const}.\text{lexval} \).

Consider the following expression:

\[
\text{let } x = 1 \text{ in let } y = (x+5) \text{ in let } x = (x+y) \text{ in } (y+x) \text{ end end end}
\]

The value of this expression is 13. Consider the \( \langle E \rangle \) parse tree node whose subtree forms the innermost \text{let} subexpression. According to the evaluation rules presented in class, the value of env for that \( \langle E \rangle \) node is a map \{x \mapsto 1, y \mapsto 6\}. The second \( \langle E \rangle \) child of that node has a subtree that corresponds to \((y+x)\). The value of env for that node is \{x \mapsto 7, y \mapsto 6\}. (Note: \( a \mapsto b \) is standard math notation for “\( a \) maps to \( b \)”.)

Try this at home: suppose we use a naive evaluation strategy in which every \( \langle E \rangle \) node has a completely separate map. What is the total number of maps and what is the total number of key-value pairs in these maps?

Next, consider a more intelligent implementation in which maps are shared for the following cases: (1) for \( (E)_1 ::= (\langle E \rangle_2 + \langle E \rangle_3) \), the two subexpressions just point to the map for the parent; (2) for \( (E)_1 ::= \text{let } \langle I \rangle = \langle E \rangle_2 \text{ in } \langle E \rangle_3 \text{ end} \), node \( \langle E \rangle_2 \) just points to the map for the parent. What is the total number of maps and what is the total number of key-value pairs in these maps?