Procedures - Building the Syntax Tree

Lecture 20
Sections 7.1 - 7.5

Robb T. Koether

Hampden-Sydney College

Wed, Apr 1, 2009
Outline

1. Function Definition Syntax Trees
   - Function Return
   - Function End

2. Function Call Syntax Trees
   - LIST Trees
   - Left- vs. Right-Recursion

3. Assignment
Building the Function Definition Syntax Tree

- To write the assembly code for the beginning of a function definition, we must know:
  - The name of the function.
  - The size of the local variable block.
- The relevant production is:
  \[ fbeg \rightarrow\] \[\text{name fargs }\] \[\{ dcls \] \[\]

Therefore, the information is available.

This information will be stored in the left and right subtrees of a \texttt{FUNC tree}. 
Building the Function Definition Syntax Tree

- From the production

\[ fbeg \rightarrow \text{name} \ \text{fargs} \ \{ \ \text{dcls} \]  

we see that the function name, parameters, and declarations have already been processed.

- Also, the function name should be global, while the parameters and declarations should be local.

- Therefore, we must call on `enterBlock()` *after* installing the function name, but *before* installing the parameters or local variables.
A Function Definition Syntax Tree

```
FUNC
  f.mode
  ID
    f.id.name
    f.mode
  NUM
    n.num
```
A Function Definition Syntax Tree

Name of the function

FUNC
f.mode

ID
f.id.name
f.mode

NUM
n.num
A Function Definition Syntax Tree

![Diagram of a function definition syntax tree with nodes labeled 'FUNC', 'ID', and 'NUM'. The 'FUNC' node is at the top, with 'ID' and 'NUM' nodes below it. The 'ID' node has a label 'f.id.name' with the 'f.mode' node below it, and the 'NUM' node has a label 'n.num' with the 'f.mode' node below it. There is an arrow pointing from the 'f.mode' node in the 'ID' node to the 'Return type' label. The diagram also shows the return type with the 'f.mode' label.]
A Function Definition Syntax Tree

- FUNCTION
  - ID: f.id.name
  - NUM: n.num
- Size of the local variable block

Diagram:

```
FUNC
  f.mode
  ID
    f.id.name
    f.mode
  NUM
    n.num
```

Size of the local variable block
Building the Function Return Syntax Tree

- To write the assembly code to return from a function, we must know
  - The return type.
  - The object to be returned.
- The return type will be stored in the SymbolTable class variable retType.
Function Return Syntax Trees
To write the assembly code for the end of a function definition, we do not need any special information, although we will know the name of the function.

This marks the *physical* end of the function.

A *return* statement marks a *logical* end.
Since this marks the physical end of the function, the block level should be decreased.

Be sure to call on `leaveBlock()` to do this, so that it can delete the hash table of local variables.
At the end of the function, there needs to be an automatic return, even if the programmer did not write a `return` statement.

But then, what about the return value?

Similarly, what if the programmer writes `return;` without specifying a return value when there should be one?
A Function End Syntax Tree

Function Definition Syntax Trees
Function Return
Function End

Function Call Syntax Trees
LIST Trees
Left- vs. Right-Recursion
Assignment
To write the assembly code for a function call, we need to know

- The name of the function.
- Each actual parameter.

Each actual parameter is an expression.

The parameters are arranged in a LIST tree.
A **LIST** tree is a recursive structure where the right subtree is an expression tree and the left subtree is
- An expression tree, or
- A **LIST** tree.
In the **LIST** tree, each **LIST** node, except the last, has an actual parameter in its right subtree and a **LIST** node in its left subtree.

The last **LIST** node has two parameters in its subtrees.
A \texttt{LIST} Tree of Four Parameters

- \texttt{LIST} Tree of Four Parameters
  - \texttt{LIST}
  - \texttt{param4}
  - \texttt{LIST}
  - \texttt{param3}
  - \texttt{param1}
  - \texttt{param2}
The syntax tree for the function call
\[ a = \text{sum}(10, 20, 30); \]
would be

```
Example (A LIST Tree)

ASSIGN INT
  ID PTR|INT value = "a"
CALL INT
  ID PTR|PROC|INT value = "sum"
LIST
  LIST
    LIST
      NUM INT value = 10
      NUM INT value = 20
      NUM INT value = 30
```
A \texttt{LIST} Tree of Two Parameters

\begin{itemize}
  \item Function
  \item Function Definition
  \item Syntax Trees
  \item Function Return
  \item Function End
  \item Function Call
  \item Syntax Trees
  \item \texttt{LIST} Trees
  \item \texttt{Left- vs. Right-Recursion}
  \item Assignment
\end{itemize}
A Tree of One Parameter

If there is only one parameter, then we do not need a LIST tree at all.
Building a Function Call Syntax Tree

The productions involved in function calls are

\[
expr \rightarrow \text{id} \ ( \ exprs \ ) \ | \ \text{id} \ ( \ )\\
exprs \rightarrow \ exprs \ , \ expr \ | \ expr
\]
Suppose the parameter list is $a, b, c$.

Then by the LALR algorithm,

- $a$ will match $expr$ and invoke the production
  
  \[ exprs \rightarrow expr. \]

- Then $b$ will match $expr$ and invoke the production
  
  \[ exprs \rightarrow exprs, expr. \]

- Similarly, $c$ will match $expr$ and invoke the production
  
  \[ exprs \rightarrow exprs, expr. \]
Had the production been

\[ \text{exprs} \rightarrow \text{expr} \, , \, \text{exprs}, \]

then

- a, b, and c would each match \text{expr} and \text{expr} , \text{expr} , \text{expr} would be pushed onto the stack.
- Then the parser would match \text{expr} with \text{exprs}.
- Then it would match \text{expr} , \text{exprs} with \text{exprs}, and so on.
Building a Function Call Syntax Tree

- The implication is that the expressions would be recognized as part of an actual argument in reverse order, from right to left.
- Does that matter?
- Consider the dynamics of building the AST and later traversing it.
A Function Call Syntax Tree

CALL
   f.mode

ID
   f.mode

LIST

LIST
   param3

param1

param2
A Function Call with Only One Parameter

CALL
f.mode

ID
f.mode

param1
Assignment

Homework

- Read Sections 7.1 - 7.5.