CS406: Compilers

Spring 2020

Week 6: Semantic Actions and Code Generation

Recap - Semantic Analysis of Expressions

- Fully parenthesized expression (FPE)
  - Expressions (algebraic notation) are the normal way we are used to seeing them. E.g. 2 + 3
  - *Fully-parenthesized* expressions are simpler versions: every binary operation is enclosed in parenthesis
    - E.g. (2 + (3 * 7))
    - So can ignore order-of-operations (PEMDAS rule)

Fully-parenthesized expression (FPE) – definition

- Recursive definition
  1. A number (integer in our example)
  2. *Open parenthesis* ‘(‘ followed by
     fully-parenthesized expression followed by
     an operator (‘+’, ‘-’, ‘*’, ‘/’) followed by
     fully-parenthesized expression followed by
     closed parenthesis ‘)’

Fully-parenthesized expression – notation

1. E -> INTLITERAL
2. E -> (E op E)
3. op -> ADD | SUB | MUL | DIV

A Hand-written Recursive Descent Parser for FPE

```
IsTerm(Scanner* s, TOKEN tok) { return s->GetNextToken() == tok;}

bool E1(Scanner* s) {
     return IsTerm(s, INTLITERAL);
}

bool E2(Scanner* s) { return IsTerm(s, LPAREN) && E(s) && OP(s) && E(s) && IsTerm(s, RPAREN); }

bool OP(Scanner* s) {
     TOKEN tok = s->GetNextToken();
     if((tok == ADD) || (tok == SUB) || (tok == MUL) || (tok == DIV))
               return true;
     return false;
}

bool E(Scanner* s) {
     TOKEN* prevToken = s->GetCurTokenSequence();
     if(!E1(s)) {
               s->SetCurTokenSequence(prevToken);
               return E2(s);
     }
     return true;
}
```

Start the parser by invoking `E()`.
Value returned tells if the expression is FPE or not.

Building Abstract Syntax Trees

- Can build while parsing a fully parenthesized expression

  Via bottom-up building of the tree

- Create subtrees, make those subtrees left- and right-children of a newly created root.

  Modify recursive parser:

  1. If token == INTLITERAL, return a pointer to newly created node containing a number
  2. Else
     1. store pointers to nodes that are left- and right-expression subtrees
     2. Create a new node with value = ‘OP’

Building AST Bottom-up for FPE

```
TreeNode* IsTerm(Scanner* s, TOKEN tok) {
      TreeNode* ret = NULL;
      TOKEN nxtToken = s->GetNextToken();
      if(nxtToken == tok)
                  ret = CreateTreeNode(nxtToken.val);
      return ret;
}

TreeNode* E1(Scanner* s) {
      return IsTerm(s, INTLITERAL);
}

TreeNode* E2(Scanner* s) {
      TOKEN nxtTok = s->GetNextToken();
      if(nxtTok == LPAREN) {
                  TreeNode* left = E(s);
                  if(!left) return left;
                  TreeNode* root  = OP(s);
                  if(!root) return root;
                  TreeNode* right = E(s)
                  if(!right) return right;
                  nxtTok = s->GetNextToken();
                  if(nxtTok != RPAREN); return ret;
                              //set left and right as children of root.
                  return root;
      }
```

Building AST Bottom-up for FPE…

```
TreeNode* OP(Scanner* s) {
      TreeNode* ret = NULL;
      TOKEN tok = s->GetNextToken();
      if((tok == ADD) || (tok == SUB) || (tok == MUL) || (tok == DIV))
            ret = CreateTreeNode(tok.val);
      return ret;
}

TreeNode* E(Scanner* s) {
      TOKEN* prevToken = s->GetCurTokenSequence();
      TreeNode* ret = E1(s);
      if(!ret) {
            s->SetCurTokenSequence(prevToken);
            ret = E2(s);
      }
      return ret;
}
```

Start the parser by invoking `E()`.
Value returned is the root of the AST.

Identifying Semantic Actions for FPE Grammar

- What do we do when we see a `INTLITERAL?`
  - Create a `TreeNode`
  - Initialize it with a value (string equivalent of `INTLITERAL` in this case)
  - Return a pointer to `TreeNode`

Identifying Semantic Actions for FPE Grammar

- What do we do when we see an `E (parenthesized expression)`?
  - Create an AST node with two children. The node contains the binary operator `OP` stored as a string. Children point to roots of subtrees representing `E`.