

L
e
c
t
u
r
e

Parsing and Parse Trees

- The previous lecture looked at evaluating an expression while parsing it. This lecture looks at turning an expression or sentence of a formal language into a parse tree.
- This is what most compilers do as an intermediate step towards compiling a program.
- The parse tree can then be processed. We'll see how to evaluate an expression from its parse tree.

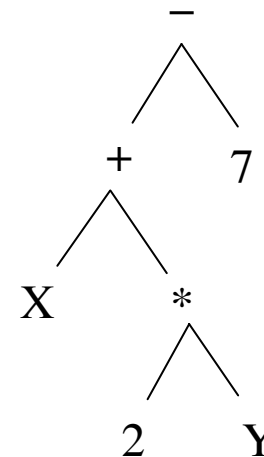
11

Parse Trees

- A parse tree is a tree structure representing sentences or expressions of a formal language that mirrors the grammar of that language, as defined by its BNF.
- For example, the parse tree for the expression

$$X + 2 * Y - 7$$

would be the following.



Arithmetic Expressions

- We're going to see how to generate the parse tree for an arithmetic expression, as defined in the previous lecture. Here's the BNF again.

```
<expression> ::= <term> |  
                <expression> + <term> |  
                <expression> - <term>  
  
<term> ::= <factor> |  
          <term> * <factor> |  
          <term> / <factor>  
  
<factor> ::= <number> | ( <expression> )
```

Parse Trees in C++

- Here's the type declaration for a parse tree for arithmetic expressions.
- The leaves of the tree (nodes with no sub-trees) are always numbers. Other nodes comprise an operator and two operands, where the operands are themselves trees.

```
class TreeNode {
public:
    bool isLeaf;        //true for leaf node
    int number;         //filled for leaf node, else
    TreeNode* leftTree; //operand 1 is another tree
    TreeNode* rightTree; //operand 2 is another tree
    char op;           // +, -, * or /
};

typedef TreeNode* TreePtr;
```

Access Routines for Parse Trees I

- We assume that we have the access routines to handle the expression string as in the last lecture (see slides 10.16 – 10.18).
- We also assume the same routines to access the parse tree:

```
bool isLeafNode (TreePtr tree);  
//returns true if tree points to leaf node  
  
int leafValue (TreePtr tree);  
//returns number at leaf node  
  
char nodeOp (TreePtr tree);  
//returns operator (+,-,*,/) at tree node  
  
TreePtr leftOf (TreePtr tree);  
//returns pointer of left sub-tree  
  
TreePtr rightOf (TreePtr tree);  
//returns pointer of right sub-tree
```

Access Routines for Parse Trees II

- In addition, we have the following two access procedures for building the parse tree and its leaves.

```
TreePtr buildLeaf (int number) {
    TreePtr newNode;

    newNode = new TreeNode;
    newNode->isLeaf = true;    //this is a leaf node
    newNode->number = number;
    newNode->leftTree = NULL;    //empty left tree
    newNode->rightTree = NULL;    //empty right tree
    newNode->op = '\0';    //no operator
    return newNode;
}
```

Access Routines for Parse Trees III

- In addition, we have the following two access procedures for building the parse tree and its leaves.

```
TreePtr buildNode (TreePtr op1, //left operand tree
                  TreePtr op2, //right operand tree
                  char op) {
    TreePtr newNode;

    newNode = new TreeNode;
    newNode->isLeaf = false; //this is NOT a leaf node
    newNode->leftTree = op1;
    newNode->rightTree = op2;
    newNode->op = op;
    newNode->number = 0; //empty number field
    return newNode;
}
```

From Expressions to Trees

- The functions that turn an arithmetic expression into a parse tree are much like the procedures in the previous lecture for evaluating an expression.
- To generate the parse tree T1 for an expression E, this is what you do.
 - Parse the next term in E, and let T1 be the resulting tree.
 - While the next character in E is an operator (“+” or “−”),
 - Read past the operator.
 - Parse the next term in E, giving tree T2.
 - Let T1 be either $\begin{array}{c} + \\ \diagup \quad \diagdown \\ T1 \quad T2 \end{array}$ or $\begin{array}{c} - \\ \diagup \quad \diagdown \\ T1 \quad T2 \end{array}$, depending on the operator.

Parsing an Expression

- Here's the C++ code.

```
void parseExpression (string& expression,
                    TreePtr& expTree) {
    TreePtr tempTree;
    char op;

    parseTerm (expression, expTree);
    while ((notEmpty(expression)) &&
           ((nextChar(expression) == '+' ) ||
            (nextChar(expression) == '-' ))) {
        op = getNextChar(expression);
        parseTerm(expression, tempTree);
        expTree = buildNode(expTree, tempTree, op);
    }
}
```

Parsing a Term

- Here's the C++ code for parsing a term. No surprises here.

```
void parseTerm (string& expression,
                TreePtr& expTree) {
    TreePtr tempTree;
    char op;

    parseFactor (expression, expTree);
    while ((notEmpty(expression) &&
           ((nextChar(expression) == '*' ||
            nextChar(expression) == '/')))) {
        op = getNextChar(expression);
        parseFactor (expression, tempTree);
        expTree = buildNode(expTree, tempTree, op);
    }
}
```

Parsing a Factor

- Here's the code for parsing a factor. Again, no surprises.

```
void parseFactor (string& expression,
                 TreePtr& expTree) {

    if (notEmpty(expression) && (nextChar(expression) != '('))
        expTree = buildLeaf(getNum(expression));
    else {
        getNextChar(expression);           // skip '('
        parseExpression(expression, expTree);
        getNextChar(expression);           // skip ')'
    }
}
```

Evaluating a Parse Tree

- Here is the routine for evaluating an arithmetic expression from its parse tree.

```
int evalTree (TreePtr expTree) {
    int result;

    if (isLeafNode(expTree))
        result = leafValue(expTree);
    else {
        switch (nodeOp(expTree)) {
            case '+':
                result = evalTree(leftOf(expTree)) + evalTree(rightOf(expTree));
                break;
            case '-':
                result = evalTree(leftOf(expTree)) - evalTree(rightOf(expTree));
                break;
            case '*':
                result = evalTree(leftOf(expTree)) * evalTree(rightOf(expTree));
                break;
            case '/':
                result = evalTree(leftOf(expTree)) / evalTree(rightOf(expTree));
                break;
            default:
                result = 0;
                cout << "Error in evaluating expression tree\n";
        }
    }
    return result;
}
```

Printing the Tree

- Here is the routine to print the tree in console window.

```
void printTree(TreePtr expTree) {  
    if (isLeafNode(expTree))  
        cout << leafValue(expTree) << endl;  
    else {  
        cout << nodeOp(expTree) << endl;  
        printTree(leftOf(expTree));  
        printTree(rightOf(expTree));  
    }  
}
```

$100 * (3 + 4) - 10 * (8 / 2)$

