NJIT

New Jersey's Science & Technology University

THE EDGE IN KNOWLEDGE

CS 280 Programming Language Concepts

About Assignment 3

Outline

- Implement a recursive descent parser
- If it is successful, do some traversals

Starter Files

- Lex.h (you can copy and use my lexical analyzer when I publish it)
- parse.h
- Partial implementations as a starting point:
  - Parsetree.SKEL.h
  - Parse.SKEL.cpp

Grammar

```
Prog := Slist
Slist := SC { Slist } | Stmt SC { Slist }
Stmt := IfStmt | PrintStmt | LetStmt |
LoopStmt
IfStmt := IF Expr BEGIN Slist END
LetStmt := LET ID Expr
LoopStmt := LOOP Expr BEGIN Slist END
PrintStmt := PRINT Expr
Expr := Prod { (PLUS | MINUS) Prod }
Prod := Rev { (STAR | SLASH) Rev }
Rev := BANG Rev | PRIMARY
Primary := ID | INT | STR | LPAREN Expr RPAREN
```


An Example Derivation

let x 3; print x;

1. Prog
2. Slist
3. Stmt SC {Slist}
4. SetStmt SC {Slist}
5. LET ID Expr SC {Slist}
6. let ID Expr SC {Slist}
7. let x Expr SC {Slist}
8. let x Prod { (PLUS|MINUS) Prod } SC {Slist}
9. let x Primary { (STAR|SLASH) Primary } { (PLUS|MINUS) Prod } SC {Slist}
10. let x ICONST { (STAR|SLASH) Primary } { (PLUS|MINUS) Prod } SC {Slist}
11. let x 3 { (STAR|SLASH) Primary } { (PLUS|MINUS) Prod } SC {Slist}
12. let x 3 { (PLUS|MINUS) Prod } SC {Slist}
13. let x 3 SC {Slist}
14. let x 3 ; {Slist}
15. let x 3 ; Stmt SC {Slist}
16. let x 3 ; PrintStmt SC {Slist}
17. let x 3 ; PRINT Expr SC {Slist}

An Example Derivation (cont)

18. let x 3 ; print Expr SC {Slist}
19. let x 3 ; print Prod { (PLUS|MINUS) Prod } SC {Slist}
20. let x 3 ; print Primary { (STAR|SLASH) Primary } { (PLUS|MINUS) Prod } SC {Slist}
21. let x 3 ; print ID { (STAR|SLASH) Primary } { (PLUS|MINUS) Prod } SC {Slist}
22. let x 3 ; print x { (STAR|SLASH) Primary } { (PLUS|MINUS) Prod } SC {Slist}
23. let x 3 ; print x { (PLUS|MINUS) Prod } SC {Slist}
24. let x 3 ; print x SC {Slist}
25. let x 3 ; print x ; {Slist}
26. let x 3 ; print x ;

Recursive Descent Parser

- One function per rule
- Function recognizes the right hand side of the rule
- If the function needs to read a token, it can read it using getNextToken()
- If the function needs a nonterminal symbol, it calls the function for that nonterminal symbol.

Token Lookahead

- Remember our lecture about wanting at most one token worth of lookahead?
- We're going to need to provide a mechanism for either "peeking" at a token or "pushing back" a token
- Easiest way to do this is to provide functions that call the existing getNextToken and add the pushback functionality
- This is called a "wrapper"

Wrapper for lookahead (given)

```
class GetToken {
      static bool pushed_back;
      static Token pushed_token;

public:
      static Token Get(istream& in, int& line) {
            if( pushed_back ) {
                  pushed_back = false;
                  return pushed_token;
            }
            return getNextToken(in, line);
      }

      static void PushBack(Token& t) {
            if( pushed_back ) {
                  throw std::logic_error("Cannot push
back more than one token!");
            }
            pushed_back = true;
            pushed_token = t;
      }
};
```

- To get a token:
  `GetToken::Get(in, line)`
- To push back a token:
  `GetToken::PushBack(t)`
  - NOTE after push back, the next time you call GetToken::Get(), you will retrieve the pushed-back token
  - NOTE an exception is thrown if you push back more than once

Parser Functions

- Each function takes a reference to an input stream and a line number
- In the event of an error, function returns 0 (a null pointer)
- If successful, the function creates a new parse tree node and returns it to the caller
- Each newly created parse tree node may point to other nodes

parse.h

```
/*
 * parse.h
 */

#ifndef PARSE_H_
#define PARSE_H_

#include <iostream>
using namespace std;

#include "lex.h"
#include "parsetree.h"

extern ParseTree *Prog(istream& in, int& line);
extern ParseTree *Slist(istream& in, int& line);
extern ParseTree *Stmt(istream& in, int& line);
extern ParseTree *IfStmt(istream& in, int& line);
extern ParseTree *LetStmt(istream& in, int& line);
extern ParseTree *PrintStmt(istream& in, int& line);
extern ParseTree *LoopStmt(istream& in, int& line);
extern ParseTree *Expr(istream& in, int& line);
extern ParseTree *Prod(istream& in, int& line);
extern ParseTree *Rev(istream& in, int& line);
extern ParseTree *Primary(istream& in, int& line);

#endif /* PARSE_H_ */
```


Parse Tree Nodes

- Each node in the tree represents what was parsed
- The children of the node are the items associated with the operation
- Example: a node representing addition would have two children, one child for each operand
- Example: a node representing Print would have one child representing the expression to print

Example: PrintStmt function

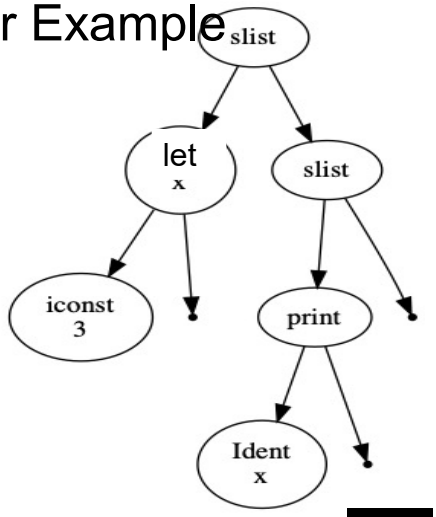
- Parser function for a Print statement has to recognize the keyword “print” (checked by getting the next token) followed by an Expr (checked by calling Expr() function).
- If the PRINT token is missing, or it is not followed by an Expr, the function fails
- If the PRINT token is present, and it is followed by an Expr, the function would make a new node for the Print; it would point to the expr to print.

Building Trees

- We use a binary tree for our parse tree
  - The base class is ParseTree
  - Derived classes for all items that need to be represented
- Each node will eventually have a type and a value
- Leaves of a parse tree are tokens
- Binary operations (such as +) are represented by having the operands as children of the node that represents the operation

Parse Tree for our Example

let x 3; print x;



tree.h and parse.cpp

- Partial implementation is given
- You will need to fill in the rest

ParseTree

```
class ParseTree {
protected:
        int             linenum;
        ParseTree       *left;
        ParseTree       *right;


public:
        ParseTree(int linenum, ParseTree *l = 0, ParseTree *r = 0)
                : linenum(linenum), left(l), right(r) {}


        virtual ~ParseTree() {
                delete left;
                delete right;
        }


        int GetLinenum() const { return linenum; }
```

IConst

```
class IConst : public ParseTree {
      int val;

public:
      IConst(Token& t) : ParseTree(t.GetLinenum()) {
            val = stoi(t.GetLexeme());
      }
};
```


Addition

```
class Addition : public ParseTree {
public:
   Addition(int line, ParseTree *l, ParseTree *r)
      : ParseTree(line,l,r) {}
};
```


Example: Prog (first rule)

```
// Prog is an Slist
ParseTree *Prog(istream& in, int& line)
{
        ParseTree *sl = Slist(in, line);

        if( GetToken::Get(in, line) != DONE )
                ParseError(line, "Unrecognized statement");


        if( sl == 0 )
                ParseError(line, "No statements in program");


        if( error_count )
                return 0;


        return sl;
}
```

- If Slist succeeds, AND all the input has been consumed, AND there's no error, return the slist parse tree
- Otherwise… error, return a null pointer

StmtList class

```
class StmtList : public ParseTree {

public:
         StmtList(ParseTree *l, ParseTree *r) : ParseTree(0, l, r) {}

};
```

- StmtList represents the list of statements with a binary tree

Slist example

```
// SC { Slist } | Stmt SC { Slist }
ParseTree *Slist(istream& in, int& line) {
     ParseTree *s = Stmt(in, line);
     if( s == 0 )
          return 0;

     return new StmtList(s, Slist(in,line));
}
```


Example: Parsing Expr

```
ParseTree *Expr(istream& in, int& line) {
        ParseTree *t1 = Prod(in, line);
        if( t1 == 0 ) {
                return 0;
        }

        while ( true ) {
                Token t = GetToken::Get(in, line);

                if( t != PLUS && t != MINUS ) {
                        GetToken::PushBack(t);
                        return t1;
                }

                ParseTree *t2 = Prod(in, line);
                if( t2 == 0 ) {
                        ParseError(line, "Missing expression after operator");
                        return 0;
                }

                if( t == PLUS )
                        t1 = new Addition(t.GetLinenum(), t1, t2);
                else
                        t1 = new Subtraction(t.GetLinenum(), t1, t2);
        }
}
```

Tree Traversals

- Postorder traversal:
  - "visit the left child"
  - "visit the right child"
  - "visit the node"

Example: node counter

```
int
ParseTree::NodeCount() const {
      int count = 0;
      if( left )
            count += left->NodeCount();
      if( right )
            count += right->NodeCount();
      return count + 1;
}
```

- Recursive
- Implements postorder traversal
- Makes sure pointers are valid before using them

NJIT New Jersey's Science & Technology University
COLLEGE OF COMPUTING SCIENCES


NJIT
THE EDGE IN KNOWLEDGE