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: "skeleton" files

Grammar

- Prog := Sl
- Sl := SC { Sl } | Stmt SC { Sl }
- Stmt := PrintStmt | PrintlnStmt | RepeatStmt | Expr
- PrintStmt := PRINT Expr
- PrintlnStmt := PRINTLN Expr
- RepeatStmt:= Repeat Expr BEGIN Stmt END
- Expr := Sum { EQ Sum }
- Sum := Prod { (PLUS | MINUS) Prod }
- Prod := Primary { (STAR | SLASH) Primary }
- Primary := IDENT | ICONST | SCONST | LPAREN Expr RPAREN

An Example Derivation

x = 3 + 3; println x;

1. Prog
2. Sl
3. Stmt SC {Sl}
4. Sum { EQ Sum } SC {Sl}
5. Prod { (PLUS|MINUS) Prod } { EQ Sum } SC {Sl}
6. Primary { (STAR|SLASH) Primary } { (PLUS|MINUS) Prod } { EQ Sum } SC { Sl }
7. IDENT { (STAR|SLASH) Primary } { (PLUS|MINUS) Prod } { EQ Sum } SC { Sl }
8. IDENT { (PLUS|MINUS) Prod } { EQ Sum } SC { Sl }
9. IDENT { EQ Sum } SC { Sl }
10. IDENT EQ Sum { EQ Sum } SC { Sl }
11. IDENT EQ Prod { (PLUS|MINUS) Prod } { EQ Sum } SC { Sl }
12. IDENT EQ Primary { (STAR|SLASH) Primary } { (PLUS|MINUS) Prod } { EQ Sum } SC { Sl }
13. IDENT EQ ICONST { (STAR|SLASH) Primary } { (PLUS|MINUS) Prod } { EQ Sum } SC { Sl }
14. IDENT EQ ICONST { (PLUS|MINUS) Prod } { EQ Sum } SC { Sl }
15. IDENT EQ ICONST PLUS Prod { (PLUS|MINUS) Prod } { EQ Sum } SC { Sl }
16. IDENT EQ ICONST PLUS Primary { (STAR|SLASH) Primary } { (PLUS|MINUS) Prod } { EQ Sum } SC { Sl }
17. IDENT EQ ICONST PLUS ICONST { (STAR|SLASH) Primary } { (PLUS|MINUS) Prod } { EQ Sum } SC { Sl }
18. IDENT EQ ICONST PLUS ICONST { (PLUS|MINUS) Prod } { EQ Sum } SC { Sl }
19. IDENT EQ ICONST PLUS ICONST { EQ Sum } SC { Sl }
20. IDENT EQ ICONST PLUS ICONST SC { Sl }

An Example Derivation (cont)

20. IDENT EQ ICONST PLUS ICONST SC { SI }
21. IDENT EQ ICONST PLUS ICONST SC SI { SI }
22. IDENT EQ ICONST PLUS ICONST SC Stmt SC { SI }
23. IDENT EQ ICONST PLUS ICONST SC PRINTLN Expr SC { SI }
24. IDENT EQ ICONST PLUS ICONST SC PRINTLN Sum { EQ Sum } SC { SI }
25. IDENT EQ ICONST PLUS ICONST SC PRINTLN Prod { (PLUS|MINUS) Prod } SC { SI }
26. IDENT EQ ICONST PLUS ICONST SC PRINTLN Primary { (STAR|SLASH) Primary } { (PLUS|MINUS) Prod } SC { SI }
27. IDENT EQ ICONST PLUS ICONST SC PRINTLN IDENT { (STAR|SLASH) Primary } { (PLUS|MINUS) Prod } SC { SI }
28. IDENT EQ ICONST PLUS ICONST SC PRINTLN IDENT { (PLUS|MINUS) Prod } SC { SI }
29. IDENT EQ ICONST PLUS ICONST SC PRINTLN IDENT SC { SI }
30. IDENT EQ ICONST PLUS ICONST SC PRINTLN IDENT SC

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)

```
namespace Parser {
bool pushed_back = false;
Tok   pushed_token;

static Tok GetNextToken(istream& in, int& line) {
      if( pushed_back ) {
            pushed_back = false;
            return pushed_token;
      }
      return getNextToken(in, line);
}

static void PushBackToken(Tok& t) {
      if( pushed_back ) {
            abort();
      }
      pushed_back = true;
      pushed_token = t;
}

}
```

- To get a token:

  `Parser::GetNextToken(in, line)`

- To push back a token:

  `Parser::PushBackToken(t)`

  - NOTE after push back, the next time you call Parser::GetNextToken(), 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). YOU NEED TO CHECK FOR THIS ERROR
- 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 "pt.h"

extern Pt *Prog(istream& in, int& line);
extern Pt *Sl(istream& in, int& line);
extern Pt *Stmt(istream& in, int& line);
extern Pt *PrintStmt(istream& in, int& line);
extern Pt *PrintlnStmt(istream& in, int& line);
extern Pt *RepeatStmt(istream& in, int& line);
extern Pt *Expr(istream& in, int& line);
extern Pt *Sum(istream& in, int& line);
extern Pt *Prod(istream& in, int& line);
extern Pt *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

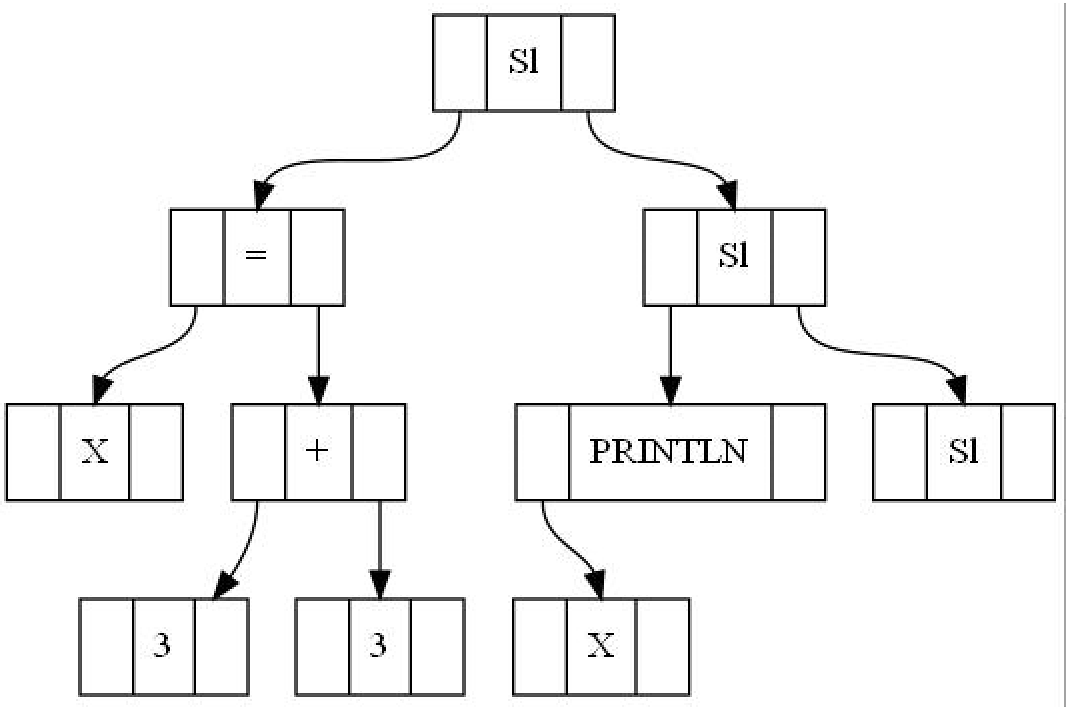
- Parser function for a Print statement has to recognize the keyword “print” (checked by getting the next token) followed by an Expr (checked by calling Expr() function).
- If the PRINT token is missing, or it is not followed by an Expr, the function fails
- If the PRINT token is present, and it is followed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

x = 3+3; println x;



tree.h and parse.cpp

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

ParseTree

```
class Pt {
          int       linenum;
          Pt        *left;
          Pt        *right;

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

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

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

IConst

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

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

Multiplication

```
class TimesExpr : public Pt {
public:
      TimesExpr(int line, Pt *l, Pt *r) :
            Pt(line,l,r) {}
};
```

Example: Prog (first rule)

```
Pt *Prog(istream& in, int& line)
{
        Pt *sl = Sl(in, line);

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

        if( error_count )
                return 0;

        return sl;
}
```

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

Sl class – Statement List

```
class Sl : public Pt {

public:
        Sl(Pt *l, Pt *r) : Pt(0, l, r) {}
};
```

- Sl represents the list of statements with a binary tree

Sl example

```
// Sl is a Stmt followed by a Sl
Pt *Sl(istream& in, int& line) {
    Pt *s = Stmt(in, line);
    if( s == 0 )
        return 0;

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

Example: Parsing Expr

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

	while ( true ) {
		Tok t = Parser::GetNextToken(in, line);

		if( t != PLUS && t != MINUS ) {
			Parser::PushBackToken(t);
			return t1;
		}

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

		if( t == PLUS )
			t1 = new PlusExpr(t.GetLinenum(), t1, t2);
		else
			t1 = new MinusExpr(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

THE EDGE IN KNOWLEDGE