Programming Project #3: Syntax Analysis

Due Date: Tuesday, October 25, 2005, Noon

Overview

Write a recursive-descent parser for the PCAT language. Section 12 of the PCAT manual gives a context-free grammar for the language to be accepted. Your program will use the Lexer you created in project 2.

Files

The following files can be found via the class web page or FTPed from:

```
~harry/public_html/compilers/p3
```

Main.java
This file has been altered to create a **Parser** object and to invoke the **parseProgram** method to parse a single PCAT program.

Token.java
StringTable.java
These files are unchanged.

Lexer.class
This is my version of the Lexer class. If you finished **Lexer.java** with no problems, you should be able to use your code; if not you may just use my Lexer. In any case, be sure you test your Parser with my Lexer since we will use my Lexer (not yours) when testing your Parser.

ParserStarter.java
This is a "starter" file which you may use to get started if you wish.

makefile
This file has been altered slightly.

tst
This is a subdirectory containing a number of test files. All the tests in this directory are syntactically legal, so they should not cause any error messages.

tst2
This is a subdirectory containing more tests. Almost all of the tests in this file are syntactically incorrect, so they should cause error messages.

go
This is a shell script that you may use to run your program on a test from **tst**. It is slightly altered from before.

run
This is a shell script that will run your program on a test in **tst**. It is unchanged.

runAll
This is a shell script that will run your program on all the tests in **tst**. It has been changed.

go2
run2
runAll2
These are versions for tests in **tst2**. These are new files.

Main.jar
This is a "black box" solution. You can execute it with a command such as:

```
java -classpath Main.jar Main tst/test1.pcat
```

This is the code that was used to produce the desired output files in the **tst** and **tst2** directories.

Do not modify any of the files except the file you are creating (**Parser.java**). In particular, DO NOT MODIFY **Main.java**, **StringTable.java**, or **Token.java**.

Comments

- Begin by looking carefully at the context-free grammar of PCAT and modifying it as necessary to accommodate a recursive-descent parser. For example, you will need to eliminate left-recursion.

- If you did not complete project #2, you may use my lexical analyzer. The compiled version is in **Lexer.class**. If you use the **makefile** provided, you will need to comment out the following lines using a **#** as shown below. (These lines call **javac** to produce **Lexer.class** from **Lexer.java**.)

```
#Lexer.class: Lexer.java
#       $(JJ) $(JFLAGS) Lexer.java
```

- The file **ParserStarter.java** is included to help you get going. It contains several auxiliary routines, which should come in handy during parsing:

```
void scan ()
void syntaxError (String msg)
void mustHave (int expectedToken, String msg)
```

Scan() can be called to advance **nextToken**.

SyntaxError() can be called to print an error message and then halt the program. For example:

```
      if (nextToken != Token.ID) {
          syntaxError ("Expecting formal parameter name");
      }
      scan ();
```

might print the following, before terminating the parser:

```
Syntax error on line 2: Expecting formal parameter name
```

MustHave() can be called to make sure that the next token is what is expected. If there is a problem, it calls **syntaxError()**. For example:

```
mustHave (Token.SEMI, "Expecting ';' after END in IF statement");
```

- I recommend against using **mustHave** for IDs, INTEGERs, REALs, and STRINGs. The reason is that you'll need to pick up the attribute information before scanning past the token.

- Please make your output match mine exactly, character for character. Although this may seem petty, it will make it easy for you (and the grader!) to run the test programs. If you make a minor change to a working program, you can use **runAll** and **runAll2** to run all of the test files. If the outputs always match exactly, then it will be easy to make sure that your minor change did not cause unexpected bugs.

Syntax Errors

PCAT programs may contain syntax errors and your parser must be capable of determining whether a PCAT program is syntactically correct or not. Your code should terminate immediately upon encountering the first syntax error, after printing an informative message stating the nature of the error. (The **syntaxError()** and **mustHave()** method will do this.)

Below is a list of the error messages that the black box solution produces. Your program should produce exactly the same messages, to make comparison of the output files easier.

```
Expecting '(' after ID in procedure definition
Expecting '(' after READ
Expecting '(' after WRITE
Expecting ')' in expression
Expecting ')' or ',' or expression in WRITE statement
Expecting ')' or l-value in READ statement
Expecting ',' or '>]' after expr in array constructor
Expecting ': type' after field name
Expecting ': type' after formal parameter name
Expecting ':= expr' after ID in record constructor
Expecting ':= expr' after ID in record constructor
Expecting ':=' after l-value in assignment stmt
Expecting ':=' in FOR statement following index var
Expecting ':=' or ':' or ',' in var definition
Expecting ';' after ')' in READ statement
Expecting ';' after ')' in WRITE statement
Expecting ';' after END in FOR statement
Expecting ';' after END in IF statement
Expecting ';' after END in LOOP statement
Expecting ';' after END in WHILE statement
Expecting ';' after EXIT statement
Expecting ';' after assignment statement
Expecting ';' after expr in RETURN statement
Expecting ';' after initializing expr in var definition
Expecting ';' after procedure body
Expecting ';' after procedure call statement
Expecting ';' after program body
Expecting ';' after type
Expecting ';' after type in type definition
Expecting ';' or ')' after type
Expecting ';' or '}' after 'ID := expr' in record constructor
Expecting ']' after array index
Expecting '}' after '}' in array constructor
Expecting 'ID := expr' in record constructor
Expecting ARRAY or RECORD
Expecting DO in WHILE statement following expr
Expecting DO or BY in FOR statement
Expecting ELSEIF, ELSE, or END in IF statement
Expecting END after statements in LOOP statement
Expecting END following statements in WHILE statement
Expecting END in FOR statement
Expecting EOF after PROGRAM IS body
Expecting ID after comma in VAR decl
Expecting ID after PROCEDURE
Expecting ID after TYPE
Expecting ID after VAR
Expecting IS after ID in type definition
Expecting IS after PROGRAM
Expecting IS in procedure definition
Expecting LValue
Expecting OF after ARRAY
Expecting PROGRAM
Expecting THEN after ELSEIF expr in IF statement
Expecting THEN in IF statement following expr
Expecting TO in FOR statement following expr-1
Expecting TypeName
Expecting VAR, TYPE, PROCEDURE, or BEGIN
Expecting a field name or END
Expecting expr or ',' or ')' in argument list
Expecting expression
Expecting field name after '.'
Expecting field name after RECORD
Expecting formal parameter name
Expecting formal parameter name
Expecting statement or END
```

Program Output

To test that your parser works correctly on programs that have no syntax errors, please insert calls to **println** your code as follows. Print each item on a single line with no spaces.

- After a string token is parsed, print the string surrounded by double quotes.
- After an identifier is parsed, print the identifier.
- After an integer token is parsed, print the integer.
- After a real token is parsed, print the value.
- After an expression using a binary operator, print the operator in postfix notation. For example, the following PCAT source:
  ```
  (1 + x) mod (1.5 <= y)
  ```
  should cause your parser to print:
  ```
  1
  x
  PLUS
  1.5
  y
  LEQ
  ```

```
        MOD
```
(Of course this source is not semantically correct, but we aren't checking that yet!)

- After an expression using a unary operator, print **unary-PLUS**, **unary-MINUS**, or **NOT**. The source:
```
        -(+(not x))
```
would print as:
```
        x
        NOT
        unary-PLUS
        unary-MINUS
```
- For assignment statements, print ASSIGN after scanning the ":=" token. Print ENDASSIGN after scanning the final ";". For example, the source:
```
        x := y + 123;
```
would print as:
```
        x
        ASSIGN
        y
        123
        PLUS
        ENDASSIGN
```
- For call statements, print CALL after scanning the function name ID. Print ENDCALL after scanning the final ";". For example, the source:
```
        foo (a, b, c);
```
would print as:
```
        foo
        CALL
        a
        b
        c
        ENDCALL
```
- For an "if" statement you should add code to print IF, THEN, ELSEIF, and ENDIF. For example, for this source:
```
        if xxx then
              a:=1;
        elseif yyy then
              b:=2;
        else
              c:=3;
        end;
```
you should this:
```
        IF
        xxx
        THEN
        a
        ASSIGN
        1
        ENDASSIGN
        ELSEIF
        yyy
        b
        ASSIGN
        2
        ENDASSIGN
        ELSE
        c
        ASSIGN
        3
```

```
ENDASSIGN
ENDIF
```

- For every other type of statement, add print statements at the beginning and at the end of the statement. For example, the source:

  ```
  return y + 123;
  ```

  would print as:

  ```
  RETURN
  y
  123
  PLUS
  ENDRETURN
  ```

  For each statement, add print statements to print out the following lines...

  ```
  ASSIGN...ENDASSIGN
  CALL...ENDCALL
  IF...THEN...ELSEIF...ELSE...ENDIF
  RETURN...ENDRETURN
  READ...ENDREAD
  WRITE...ENDWRITE
  WHILE...DO...ENDWHILE
  LOOP...ENDLOOP
  FOR...TO...BY...DO...ENDFOR
  EXIT
  ```

- After a variable declaration, print **VAR**. Print it after scanning the ";".
- After a type definition, print **TYPE**. Print it after scanning the ";".
- While parsing a procedure definition, print **PROCEDURE**. Print after the return type and before the procedure body.
- When parsing a function call in an expression, print out **FUNCTION** and **ENDFUNCTION**. For example, the source

  ```
  x := foo (123);
  ```

  would print out as:

  ```
  x
  ASSIGN
  foo
  FUNCTION
  123
  ENDFUNCTION
  ENDASSIGN
  ```

- While parsing the formal parameters in a procedure definition, print **FORMAL**. Print just after each TypeName is parsed. For example, the source:

  ```
  procedure foo (x,y,z: Integer; a: Real) : Bool is
        begin return; end;
  ```

  results in the following output:

  ```
  foo
  x
  y
  z
  Integer
  FORMAL
  a
  Real
  FORMAL
  Bool
  PROCEDURE
  RETURN
  ENDRETURN
  ```

Testing

We will use my **Lexer.class** when testing your program, but this should not result in different output than if your Lexer is used. We will use only the tests in **tst** and **tst2**. If your program passes all of those, then it should be okay. None of the test files has lexical errors or unusual tokens, so the behavior of the Lexer should not be a large issue.

Error Recovery???

Your code should simply quit after the first syntax error is discovered, but this is not a very good solution for a production compiler. If you are a really clever programmer, you might want to try the challenge of adding *error recovery* to your parser. The idea is to patch things up after a syntax error and keep parsing.

My experience suggests that doing error recovery will double the time required to do a recursive-descent parser; do not attempt this until you have completely finished this basic parser. Please hand in only code that behaves the way the black box parser behaves. Any error recovery would be for your own education and will not be graded.

Ideas for Getting Started

There are 33 test files in the **tst** directory and 68 test files in **tst2**. Where does one begin?

These test files are designed to test working programs, but may not be the best way to develop working code. Just starting with the first file (**arrays1.pcat**) and trying to get it working is not a great idea.

Instead, remember that you can create your own test files. Also, you can type input directly into your program while it runs to see what it is doing.

When you type programs in directly, remember that your program will generally be looking at the NEXT token, so it may be waiting for a new token in an unexpected way. Just type an EOF (control-d) or some more tokens if your program seems to be hanging or has failed to detect an error. It may have just been waiting for another token before issuing its output.

One approach is to write all the parsing routines, then start debugging. A better approach is to write a little code and get it working before moving on. For example concentrate on these methods first:

parseBody
parseStmts
parseReturnStmt

So that you can test them, you'll need to create dummy "stub" methods for **parseDecls** and **parseExpr**.

parseDecls – This stub method will parse nothing at all. It will simply return.
parseExpr – This stub method will simply call

```
  mustHave (Token.INTEGER, "Expecting expression (integer only)");
```

and return.

With this framework, you should be able to parse simple PCAT programs like these:

```
    program is
    begin
        return;
    end;
```
and
```
    program is
    begin
    end;
```
and
```
    program is
    begin
        return;
        return 123;
        return;
        return 456;
    end;
```

Once you have this working, you can start to code the real **parseExpr** method and the methods it calls. Now you can test the code for parsing Expressions with a program like this:

```
    program is
    begin
        return <<any expression>> ;
    end;
```

Details...

Please email your **Parser.java** file as a plain-text attachment to:

```
   cs321-01@cs.pdx.edu
```

Don't forget to use a subject like:

```
   Proj 3 - John Doe
```

Your code should behave in exactly the same way as my code. If there is any question about the exact functionality required,

(1) Use my code (the "black box" **.jar** file) on test files of your own creation, to see how it performs.

(2) *Please ask* or talk to me!!! I will be happy to clarify any of the requirements. If there are any problems with the assignment, I would like to alert other students and/or modify my documents or files. If my test data can be improved, please let me know.

Don't submit multiple times. Be sure to keep an unmodified copy of your file on Sirius with the timestamp intact. Work independently: you must write this program by yourself.