Programming Project 1: An Interpreter for the "E" Language

Due Date: Tuesday, October 11, 2005, Noon

Overview

This project provides an overview of compiling. It will introduce you to lexical analysis, parsing, semantic checking, and interpretation. In addition, you will gain experience programming in Java.

For this assignment, you will be given a Java program which is almost complete. You will modify and add to this program.

The "E" Language

The "E" language is a tiny, toy programming language. It has variables, assignment statements, if-then-else statements, and simple expressions. The variables have integer values. Here is a context-free grammar (CFG) giving the syntax of the "E" language.

Program	→ Expr
Expr	→ Term
	→ Term '+' Term
	→ Term '-' Term
	→ **set** ID '=' Expr
	→ **if** Expr **then** Expr **else** Expr
Term	→ ID
	→ NUM
	→ '(' Expr ')'
	→ '{' Block '}'
Block	→ **var** VarList ';' ExprList
	→ ExprList
VarList	→ ID
	→ ID VarList
ExprList	--> Expr
	--> Expr ';' ExprList

Here is an example "E" program:

```
{ var x y z;
  set x = 100;
  { var a;
    set a = x + 3;
    a + x
  };
  set y = x + (z + (3+x));
  x
}
```

The keywords are **var**, **set**, **if**, **then**, and **else**. Identifiers (IDs) are sequences of alphanumeric characters, beginning with a letter, and excluding keywords. Numbers (NUMs) are sequences of decimal digits.

A program consists of a single expression. When the program is executed, the expression is evaluated and the result is printed.

The semantics of the language are given informally next:

- A Term consisting of an identifier "x" evaluates to the current value of variable "x".
- A numeric Term "n" evaluates to the integer represented by the digits of "n".
- The expression "$Expr_1$ + $Expr_2$" is evaluated by first evaluating the left-hand sub-expression "$Expr_1$" and then evaluating the right-hand sub-expression "$Expr_2$" and then adding the two values. The expression "$Expr_1$ - $Expr_2$" is evaluated similarly.
- The expression "**set** ID = Expr" evaluates the right-hand side "Expr" and then assigns that value to the variable ID on the left-hand side. The value of the overall set expression is the new value of the variable.
- The Term "(Expr)" has the value of "Expr".
- A Block has the general form "{ **var** x_1 x_2 ... x_N ; $Expr_1$; $Expr_2$; ... ; $Expr_M$ }". It introduces the variables x_1, x_2, ..., x_N with initial values of zero. The scope of the variables is the list of expressions "$Expr_1$; $Expr_2$; ...; $Expr_M$". Within these expressions, any outer declarations of any of the variables x_1, x_2, ..., x_N are hidden. All variables must be declared in this manner before being referenced or assigned to. The block is evaluated by first assigning an initial value of zero to all the newly introduced variables, then by executing each of $Expr_1$; $Expr_2$; ... ; $Expr_M$ in turn. The overall value of the block is the value of the last expression, $Expr_M$.
- The expression "**if** $Expr_1$ **then** $Expr_2$ **else** $Expr_3$" is evaluated as follows: First $Expr_1$ is evaluated. If the value is 1 then $Expr_2$ is evaluated and its value serves as the value of the entire expression. If the value of $Expr_1$ is not 1, then $Expr_3$ is evaluated and its value serves as the value of the entire expression.

The Assignment

Your task is to produce a Java program that reads in an "E" program, parsing it and checking it for correctness. If the program contains no errors, your program will then interpret the "E" program (that is, it will execute the "E" program according to the semantics given above) and will print the result of its execution as a single integer.

Your solution must consist of a single file called

```
Proj1.java
```

which will contain a class called "Proj1" with a method

```
public static void main (String [] args)
```

Your program should expect a single command-line argument specifying the name of a file containing the "E" program to interpret. If the "E" program parses and checks correctly, the interpreted result should be written to standard output. Any parsing or checking errors should be written to standard error. Don't worry about the possibility of integer overflow during execution.

Implementation

Much of the work has already been done and is provided for you in a "starter" file called:

```
Proj1Starter.java
```

which can be accessed through the class web page, or at:

```
~harry/public_html/compilers/p1/
```

Begin by creating a directory for this project (perhaps called "p1"). Then, get a copy of the starter file. Compile and execute it, with commands such as these (shown for Unix):

```
% vi Proj1.java  -or-  % emacs Proj1.java
...
% javac Proj1.java
% java Proj1 test.e
```

This file already contains the code to scan, parse, and check an "E" program. It does not contain any code to interpret the program; you'll have to add that. Also, the starter file does not handle the subtraction operator "-" or the "if-then-else" expression; you'll have to add that, too. You are strongly encouraged to use this file as a starting point, but it is not required.

How the Starter Program Works

The starter file (Proj1Starter.java) contains the following classes:

```
Proj1
Token
Lexer
Expr
  VarExpr
  ConstExpr
  PlusExpr
  SetExpr
  BlockExpr
Env
Parser
```

The indentation shows that some of the classes are subclasses of "Expr". Normally, each class should go into a different file, but for a small program like this, it makes sense to place all classes in one file.

The Proj1 class contains just a single method called "main", which provides the overall control sequence of the program.

We begin by breaking the input file into tokens. Each token is described by an integer, known as the token type. This describes the kind of token we have, such as ID, NUM, LPAREN, etc. For some tokens, there is additional information (stored in the variable "attribute"), such as a String giving the characters in an identifier, or the integer value of a number.

The class Token contains a number of useful definitions, but is never instantiated. In other words, no Token objects are ever created. The class Token only contains constants such as ID, NUM, and LPAREN, etc.

There will be a single instance of class Lexer, called "lexer". This object does the scanning of the input file. Whenever we need a new token, we will send the lexer object the "getToken" message:

```
tok = lexer.getToken ();
```

This method will return the type of the next token and, for some kinds of tokens, it will set the "attribute" variable as well. (The variable "attribute" is a field in the Lexer object.)

The class "Expr" and its subclasses are used to construct the Abstract Syntax Tree (AST) which will represent the program. The goal of parsing is to read tokens and create an Abstract Syntax Tree. The tree has several different kinds of nodes. For example, a leaf representing a variable will be represented by a "VarExpr" object. An interior node representing an addition expression will be represented by a "PlusExpr" object, which will contain pointers to two sub-trees representing the two operands.

We will need to keep track of which variables have been declared and we will do this with a linked list. This linked list is called the "environment". Each element in the list will be represented with an "Env" object, which contains a "next" pointer to another Env object, as well as a String "key".

```
class Env {
  String key;
  Env next;
  ...
}
```

One method for class Env is called “find”; it will search the list looking for a given String. Another method is called “toString”; it will produce a printed version of the entire list.

The parsing is done by the routines in the class “Parser” named:

```
parseProgram
parseId
parseExpr
parseTerm
```

Each of these will return an AST representing the thing parsed. For example, “parseExpr” will return an AST representing an Expression. The method called “parseProgram” will create the Lexer object and will parse the entire program. The parsing methods are recursive, calling each other in a way that parallels the context-free grammar of the “E” language.

The “Parser” class is never instantiated; no Parser object is ever created. Instead, the parsing methods are all static methods in class Parser. In addition, there are some helper methods used while parsing:

```
syntaxError
scan
mustHave
```

The Parser class has a couple of static variables:

`lexer` – A pointer to the Lexer object
`tok` – The type of the current token (e.g., ID, NUM, LPAREN, ...)

There is a class called “CompileTimeError,” which is a kind of Exception. An instance of this class will be created whenever a CompileTimeError is thrown. When the exception is thrown, it will be created by passing line number where the problem occurred and a textual description of the error. This exception will be caught and handled by printing the message in the following format and terminating the program.

`Error on line` *...line number..*`:` *...error message...*

When a parsing error occurs, the program will invoke the “syntaxError” method, which is passed the kind of token expected. Using this token type and the type of the current token, it will throw a CompileTimeError to print a message such as the following, and terminate the parse.

```
Error on line 7: Expecting identifier, but found number
                                                     instead!
```

The semantic checking is done by methods in the AST classes. In particular, there is a method called “check” which is implemented in each of the subclasses of Expr.

```
boolean check (Env env) {...}
```

The check method is implemented differently in each of the various AST classes. Each time the check method is invoked, it is invoked on some node in the AST. It will check the subtree rooted at that node.

The exact nature of the checking depends on the nature of the node. For example, if the subtree is a simple leaf representing a constant number, no checking is needed:

```
class ConstExpr extends Expr {
  ...
  boolean check (Env env) {
    return true;
  }
}
```

The check method is passed an environment (telling which variables are defined so far) and it returns a Boolean indicating true if the AST was checked and no errors were found. If errors are found, it prints an error message and returns false.

Hints on How to Proceed

First, read over the starter file completely and understand how it works.

Next, add the necessary code to handle the "-" operator. Create a new subclass of "Expr". Add code to parse a subtraction expression and return an instance of this new class. Make sure that the "check" and "toString" methods in the new class work properly.

Next, add the necessary code to handle the "if-then-else" construct. Create a new subclass of "Expr".

Next, create a subclass of class "Env" (perhaps you will call it "ValueEnv") that will add a "value" field. A ValueEnv is a linked list of key-value pairs, where the key is a String and the value is an integer. A ValueEnv will be used to hold the current variables' values. [Alternatively, you might choose to just modify the class Env, instead of creating a subclass.]

Add a new method called "eval" to class Expr and to each of its subclasses:

```
int eval (ValueEnv env)
```

This method is passed a ValueEnv giving the current values of the variables. This method will walk the AST as necessary and return the result of evaluating each subtree. The "main" routine will only try to interpret correct programs; it will terminate without calling "interpret" if there are semantic errors. Therefore, you can assume within "eval" that the AST contains no errors.

The "eval" method will be structured very similarly to the "check" method, and will walk the AST in the same way.

What is Required

A working solution to this assignment is on the web page in the file:

```
Proj1.jar
```

To run this program on an "E" source file "foo.e", download it and execute:

```
java –classpath Proj1.jar Proj1 foo.e
```

Your program should generate exactly the same output as this one. Error messages should also be identical.

The following files are provided to help you in debugging your code. These files can help you run your program against a set of "test files". You are free to modify and use these files in any way you wish; they are not part of the assignment.

`tst`	A subdirectory containing the test data
`testa.e`	A sample test file (input to your program)
`testa.out.bak`	The desired output your program should produce (stdout)
`testa.err.bak`	The desired output your program should produce (stderr)
<Other test files: `testb`, `testc`, ...>	
`go`	A shell script to run a test and display the results
`run`	A shell script to run a test and display differences to the expected results
`runAll`	A shell script to run several tests at once and display differences to the expected results

These tests should not be considered complete or exhaustive. You are encouraged to create additional tests.

Assignment Submission

Place your code in a single file named "Proj1.java" and email it to:

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

You must include your file as a *PLAIN TEXT ATTACHMENT.* The contents of the email itself will be ignored, only the attachment will be used!

The subject line of the email must say "Proj 1 – Xxxxxx Yyyyyy", where "Xxxxxx Yyyyyy" is your name. Please use your full name as it appears in the PSU registration system. Both pieces of information on the subject line are important.

Example subject lines:	
`Proj 1 - John Doe`	← *Correct*
`Proj 1 - Doe`	← *Wrong: Use full, "official" name!*
`Proj 1`	← *Wrong: Missing name!*
`John Doe`	← *Wrong: Missing project number!*
	← *Wrong: Missing subject altogether!*

DO NOT SUBMIT YOUR PROGRAM TWICE!

If for any reason there is any problem or question, please email the instructor and ask. Do not simply submit a second time "just to be sure". You may only submit your code a second time after securing approval from either the instructor or the grader.

Once a second submission has been approved, use a subject line such as:

```
Proj 1 – Xxxxxx Yyyyyyy - Second submission
```

We should be able to compile your program by creating a fresh directory, saving your attachment, and typing:

```
% javac Proj1.java
```

To run your program on input file “foo.e”, we should be able to type:

```
% java Proj1 foo.e
```

We will be using automated mechanisms to read, compile, and test your programs, so adherence to this naming and mailing policy is crucial and mandatory! You will lose points if you fail to submit your program in the correct way.

Your program is due at noon. If it is emailed at 12:01 it will be considered late and you will lose 50% of the grade. If it is later than 24 hours, it will not be accepted. The reason I ask for programs at noon is because I want students to have plenty of time to get to class after submitting their programs. The reason that I take so many points off for lateness is that if I don’t, many people will ignore the deadline.

Please email yourself a copy of the program and keep that email for a while. Should there be any problems in the email system, you can always forward this email to us when requested and we’ll see the older timestamp.

Also, please keep a copy of your file on Sirius as you submitted it; do not modify this file. If any issues arise, we can also look at the timestamp on this file.

Working Together

Do not work together on any programming project in this class. Do not look at anyone else’s code. Do not allow anyone to use your code. Every line of code you submit must be your own work (except of course the code we distribute to the class). You may discuss Java and the assignments with other students, but only in general terms. You may not look at someone else’s code or share Java code. Violations to these rules constitute cheating.

Questions/Comments via Email

Questions/comments may be directed to either me or to the class grader. The best approach is through email:

`harry@cs.pdx.edu`	← *Email to the instructor*
`david.archer@dsl-only.net`	← *Email to the class grader*

If you send questions via email, please include “cs321” in the subject line so we don’t accidentally identify your email as junk!

Do not send questions to the class account:

`cs321-01@cs.pdx.edu` ← *Project submissions, only!*