CSE 3341 Project 1 - CORE Interpreter

Overview

The goal of this project is to build an interpreter for a version of the Core language discussed in class. This handout defines a variation of this language; make sure you implement an interpreter as described in this handout, not as described in the lecture notes. You must use on of these languages for the project implementation: C, C++, Java, or Python. Some constraints on your implementation:

- Do not use scanner generators (e.g. lex, flex, jlex, jflex, ect) or parser generators (e.g. yacc, CUP, ect)
- Do not use functionality from external libraries for complex text processing. However, you can use all functionality from stdlib.h/string.h or java.lang.String.

Your submission should compile and run in the standard environment on stdlinux. If you work in some other environment, it is your responsibility to port your code to stdlinux and make sure it works there. The graders will not spend any time porting your code - if it does not work on stdlinux, they will not grade it.

The syntax of the language was discussed in class. However, your implementation should be based on the following variations:

- The production for a statement is ⟨stmt⟩ ::= ⟨assign⟩ | ⟨if⟩ | ⟨loop⟩ | ⟨in⟩ | ⟨out⟩ | ⟨case⟩. Here the case statement is the one defined in assignment 2.
- The production for input is now ⟨input⟩ ::= input ⟨id-list⟩ ;.

The semantics of this modified language should be obvious; if some aspects of the language are not, please bring it up on Piazza or in class for discussion. You need to write a language for the exact language described here. Every valid input program for this language should be executed correctly, and every invalid input program for this language should be rejected with an error message.

The interpreter should have four main components: a lexical analyzer (a.k.a scanner), a syntax analyzer (a.k.a. parser), a printer, and an executor. The parser, printer and executor must be written using the recursive descent approach discussed in class.

Main

The main procedure should do the following. First, read and parse the input program. Next, invoke the printer on the parse tree. Finally, read the input values for the execution and use them in the executor to run (interpret) the input program.

Input

The input to the interpreter will come from two ASCII text files. The names of these files will be given as command line arguments to the interpreter. The first file contains the program to be executed. The second file contains the input data on which this program will be executed. Each input statement in the program will read the next data value from the second file. Since Core has only integer variable, the input values in the second file will be integers, separated by spaces and/or tabs and/or newlines. NOTE: for this project we will allow negative integers (e.g. the input file could be -2 45 0 3 -55).

The scanner should process the sequence of ASCII characters in the first file and should produce a sequence of tokens as input to the parser. The parser performs syntax analysis of this token stream. As discussed in class, getting the tokens is typically done on demand: the parser asks the scanner for the current token, or moves to the next token. There are two options for creating the token stream: (1) upon initialization, the scanner reads the entire program from the file, tokenizes it, and stores all tokens in some list or array, or (2) upon initialization, the scanner reads from the file only enough characters to construct the first token, and then later reads from the file on demand as the parser asks for the next token. Real compilers and interpreters use (2); in your implementation, you can implement (1) or (2), whichever you prefer.

The input program contains a non-empty sequence of ASCII characters. To get them, you should use the standard libraries for I/O in your chosen language. The interpreter should exit back to the OS after processing the entire sequence of input characters. If an unexpected end-of-file is encountered (e.g. in the middle of a program), and error message should be printed and the interpreter should exit back to the OS.

The first input file contains the source code of the program. For example, the input could be

```
program
int                                    x
, y, z; begin input x, y;
z:= x; x :=y; y:=
z;              output x; output y ; end
```

For this input your scanner should produce the following sequence of tokens:

PROGRAM, INT, ID[x], COMMA, ID[y], COMMA, ID[z], SEMICOLON, BEGIN, INPUT, ID[x], COMMA, ID[y], SEMICOLON, ID[z], ASSIGN, ID[x], SEMICOLON, ID[x], ASSIGN, ID[y], SEMICOLON, ID[y], ASSIGN, ID[z], SEMICOLON, OUTPUT, ID[x], SEMICOLON, OUTPUT, ID[y], SEMICOLON, END, EOF

Feel free to add additional "helper" tokens or pass along additional information from the scanner to the parser (doing this might help with catching and generating error messages, for example). The scanner is case sensitive. An ID has the following production rules:

$$\langle \text{id} \rangle ::= \langle \text{letter} \rangle \mid \langle \text{id} \rangle \langle \text{letter} \rangle \mid \langle \text{id} \rangle \langle \text{digit} \rangle$$

A keyword takes precedence over an id; for example, "begin" should produce the token

BEGIN, but "bEgIn" should produce the token ID.

Lexical Analysis

When implementing the scanner, do not use external libraries or tools that allow complex text processing. The purpose of these restrictions is to give you hands-on experience with the implementation details of lexical analysis, using only built-in functionality from mainstream languages. Some suggestions for building the scanner are included at the end of the lecture notes on grammars. If you need clarification for some scanning issued, ask a question in class or on piazza.

Syntax Representation

The parser processes the stream of tokens produced by the scanner, and builds the parse tree representation.

Parse Tree Representation

You need to implement a ParseTree data type. This data type will implement the parse tree abstraction, so that the rest of the interpreter does not have to worry about how parse trees are stored. If you ignore this issue and implement your interpreter so that components such as the executor/parser/printer are aware of the internal details of how parse trees are stored, you can expect a reduction in your project score.

To simplify life, you can assume a predefined upper limit on the number of nodes in the parse tree; 5000 should be enough. Also, you can assume that there will be no more than 1000 distinct identifiers (each of side no more than 50 characters) in any given program that your interpreter has to execute. If you feel like implementing something more realistic, remove these limits.

Output

All output should go to stdout. This includes error messages - do not print to stderr. The printer should produce "pretty" code with the appropriate indentation. To make things somewhat uniform, let's say that the different levels of indentation will be separated by two spaces. Also, use as few spaces in the output code: e.g. instead of

```
input x, y ;
z := x ;
```

use

```
input x,y;
z:=x;
```

For the executor, each output intger should be printed on a new line, without any spaces/tabs before or after it. The output for error cases is described below.

Invalid Input

Your scanner, parser, and executor should recognize and reject invalid input. Handling of invalid input is a part of the language semantics, and it will be taken into account in the grading of the project. For any error, you have to catch it and print a message. The message must have the form "ERROR: some description" (ERROR in uppercase). The description should be a few words and should accurately describe the source of the problem. You do not need to have sophisticated error messages that denote exactly where the problem is in the input file. After printing the error message to stdout, just exit back to the OS. But make sure you catch the error conditions! When given invalid input, you interpreter should not "crash" with a segmentation fault, uncaught exceptions, ect. Up to 20% of your score will depend on the handling of incorrect input, and on printing error messages exactly as specified above.

There are several categories of errors you should catch. First, the scanner should make sure that the input stream of characters represents a valid sequence of tokens. For example, characters such as '_' and '%' are not allowed in the input stream. Second, the parser should make sure that the stream of tokens is correct with respect to the context-free grammar.

In addition to scanner and parser errors, additional checks must be done to ensure that the program "makes sense" (semantic checking). In particular, after the parse tree is constructed you should make sure that every ID that occurs in the statements has been declared in the declaration part of the program (if you prefer, you can make these checks during the parsing rather than immediately after it). If we have a variable in ⟨stmt-seq⟩ that is not declared in ⟨decl-seq⟩, this is an error. Also, report as error any "doubly declared" variables, e.g."int x,y; int z,x;" in the declaration sequence should result in an error.

Your executor should also check that during the execution of the program, whenever we read the value of a variable, this variable has already been initialized. For example, if the program is

```
program
int x,y;
begin
x:=y;
end
```

the executor should complain when it tried to execute the statement x:=y, because we are trying to read the value of y and this variable has not been initialized yet. immediately after the variable declarations and before begin, all declared variables are uninitialized.

Testing Your Project

On the course web page I will post some test cases. For each test case (e.g. t4) there are three files (e.g. t4.code, t4.data, and t4.expected). For the tests containing valid inputs, you need to do something like

```
myinterpreter t4.code t4.data > t4.out; diff t4.out t4.expected
```

You should get no differences from diff - everything should be exactly the same. For the tests containing invalid inputs, something like

```
myinterpreter bad2.code bad2.data > bad2.out; more bad2.out
```

shoould show an error message "ERROR: ..." in file bad2.out.

The test cases are very, very weak. You must do extensive testing with your own test cases. Read carefully the lecture notes and be very thorough with all details.

Implementation Suggestions

There are many ways to approach this project. Here are three suggestions:

- Plan to spend a significant amount of time on the scanner and parser. Once they are working correctly, the printer and executor should be relatively straightforward.
- Pick a small subset of the language (e.g. only a few of the grammar productions and implement a fully functioning scanner and parser for that subset. Do extensive testing. Add more grammar productions. Repeat.
- Start early. The project is too complex to be completed in a few days.
- Post questions on piazza, and read the questions other students post. You may find details you missed on your own.
- You are encouraged to share test cases with the class on piazza.

Project Submission

On or before 11:59 pm Oct 4th, you should submit the following:

- One of more files for the interpreter (just the source code).
- An ASCII text file named README.txt that contains:
  - Your name on top
  - The names of all the other files yout are submitting and a brief description of each stating what the file contains
  - Instructions on how to compile and run the project (please be exact here)
  - Any special features or comments on your project
- A document file (also ASCII text). This file should contain:
  - Your name on top
  - A description of the overall design of the interpreter, including the parse tree representation (with brief description of the methods/functions used as interface between the parse tree and the rest of the system), and the interactions between the scanner and the parser (with breif description of the interface methods/functions between the two)

- A brief description of how you tested the interpreter and a list of known remaining bugs (if any)
- If you borrowed ideas or anything else (e.g. code) from anywhere, describe briefly.

Submit your project as a single zipped file to the Carmen dropbox for Project 1.

If the time stamp on your submission is 12:00 am on Oct 5th or later, you will receive a 10% reduction per day, for up to three days. If your submission is more than 3 days late, it will not be accepted and you will receive zero points for this project. If you resubmit your project, only the latest submission will be considered.

If the graders have problems compiling or executing your program, they will email you; you must respond within 48 hours to resolve the problem. Please check often your xyz.123@osu.edu account after submitting the project in case the graders need to get in touch with you.

Grading

The project is worth 100 points. Correct functioning of the interpreter is worth 65 points. The handling of error conditions is worth 20 points. The implementation style and documentation are worth 15 points.

Academic Integrity

The project you submit must be entirely your own work. Minor consultations with others in the class are OK, but they should be at a very high level, without any specific details. The work on the project should be entirely your own; all the design, programming, testing, and debugging should be done only by you, independently and from scratch. Sharing your code or documentation with others is not acceptable. Submissions that show excessive similarities (for code or documentation) will be taken as evidence of cheating and dealt with accordingly; this includes any similarities with projects submitted in previous instances of this course.

Academic misconduct is an extremely serious offense with severe consequences. Additional details on academic integrity are available from the Committee on Academic Misconduct (see http://oaa.osu.edu/coamresources.html). If you have any questions about university policies or what constitutes academic misconduct in this course, please contact me immediately.