CSE 3341 Project 2 - CORE Interpreter

Overview

The goal of this project is to build an interpreter for the Core language discussed in class. At the end of this handout is the grammar you should follow for this project. This project should be completed using C.

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 semantics of the Input and Output statements are given in the sections below. Other semantic details of this language should be for the most part obvious; if any aspects of the language are not, please bring it up on Piazza for discussion. You need to write a parser for the language described. 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

Your project should have a main procedure in a file called main.c, which does the following in order:

1. Read and parse the input program.
2. Invoke the printer on the parse tree.
3. Execute the input program.

I have provided you a main.c file which does this, but feel free to modify it if you see the need to or would like to structure the project differently from how I suggest.

Input to your interpreter

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. During execution each Core input statement in the first file will read the next data value from the second file. Since Core has only integer variables, the input values in the second file will be integers, separated by spaces and/or tabs and/or newlines. NOTE: while the language only allows positive integers as constants, the input file can have negative integers (e.g. the input file could be -2 45 0 3 -55).

The scanner processes 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. Feel free to modify your scanner from project 1 or the scanner I provide; for example, you may want to add additional "helper" tokens or create new functions to help catch and generate error messages.

Parse Tree Representation

You should generate a parse tree for the input program, using the top-down recursive descent approach described in class.

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 size no more than 10 characters) in any given program that your interpreter has to execute. If you feel like implementing something more realistic, remove these limits.

Output from your interpreter

All output should go to stdout. This includes error messages - do not print to stderr.

The parser functions should only produce output in the case of an error.

The printer functions should produce "pretty" code with the appropriate indentation, i.e. statement sequences for if/while statements should be indented, and nested statements should be further indented. I do not have specific format requirements here, the printer functions are mainly to help you verify your parse tree has correctly captured the structure of the program.

To make things somewhat uniform, please use the tab character ''$\widehat{\ }$to create indentations. 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 Core output statement should produce an integer printed on a new line, without any spaces/tabs before or after it. The output for error cases is described below. Other than that, the executor functions should only have output if there is an error.

Invalid Input

Your scanner, parser, and executor should recognize and reject invalid input. For any error, you have to catch it and print a message. The message should 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 useful error messages.

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 use 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 produce an error when it tries to execute the statement x:=y, because we are trying to read the value of y and this variable has not been initialized yet.

Testing Your Project

I will provide some test cases. The test cases I will provide are rather weak. You should do additional testing testing with your own cases. For each test case I provide (e.g. t4) there are three files (e.g. t4.code, t4.data, and t4.expected). For the tests containing valid inputs, if you compile your project with the command "gcc main.c" so it generates an executable "a.out" you need to do something like

```
./a.out t4.code t4.data > t4.out; diff t4.out t4.expected
```

You should get few differences from diff - everything should be exactly the same (slight variations in the print function are allowed). For tests containing invalid inputs,

```
./a.out bad2.code bad2.data > bad2.out; cat bad2.out
```

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

Suggestions

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

- Plan to spend a significant amount of time on the parser. Once it is 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 parser for that subset. Do extensive testing. Add more grammar productions. Repeat.
- 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.
- Note that the grammar enforces right-associativity. In Core $1 - 2 - 3 = 2$, not -4.

Project Submission

On or before 11:59 pm June 17th, you should submit the following:

- Your complete source code.
- An ASCII text file named README.txt that contains:
  - Your name on top
  - The names of all the other files you are submitting and a brief description of each stating what the file contains
  - Any special features or comments on your project
  - 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 brief 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)

Submit your project as a single zipped file to the Carmen dropbox for Project 2.

If the time stamp on your submission is 12:00 am on June 17th 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.

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.

<prog> ::= **program** <decl-seq> **begin** <stmt-seq> **end**

<decl-seq> ::= <decl> | <decl><decl-seq>

<stmt-seq> ::= <stmt> | <stmt><stmt-seq>

<decl> ::= **int** <id-list> **;**

<id-list> ::= <id**>** | <id**> ,** <id-list>

<stmt> ::= <assign> | <if> | <loop> | <in> | <out>

<assign> ::= <id> **=** <expr> **;**

<in> ::= **input** <id> **;**

<out> ::= **output** <expr> **;**

<if> ::= **if** <cond> **then** <stmt-seq> **endif ;**

| **if** <cond> **then** <stmt-seq> **else** <stmt-seq> **endif ;**

<loop> ::= **while** <cond> **begin** <stmt-seq> **endwhile ;**

<cond> ::= <cmpr> | **! (** <cond> **)**

| <cmpr> **or** <cond>

<cmpr> ::= <expr> **==** <expr> | <expr> **<** <expr>

| <expr> **<=** <expr>

<expr> ::= <term> | <term> **+** <expr> | <term> **–** <expr>

<term> ::= <factor> | <factor> ***** <term>

<factor> ::= <const> | <id> | **(** <expr> **)**

<const> ::= **0** | **1** | **2** | … | **1023**

<id> ::= <letter> | <id><letter> | <id><digit>

<letter> ::= **a** | **b** | … | **z** | **A** | **B** | **…** | **Z**

<digit> ::= **1** | **2** | … | **9**