CSE 3341: Interpreter for Core

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 one of these languages for the project implementation: C, C++, or Java. Some constraints on your implementation:

- Do not use scanner generators (e.g., lex, flex, jlex, jflex) or parser generators (e.g., yacc, CUP).
- Do not use functional languages such as Scheme or Lisp.
- 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. For C and C++, use the standard compilers on stdlinux. For Java, use the current JDK version on stdlinux (to get it, you need to run **subscribe**, subscribe to JDK-CURRENT, log out, log in again). If you work in some other environment, it is your responsibility to port your code to stdlinux (well in advance of the deadline) and to make sure that it works there. **The grader will not spend any time porting your code – if it does not work on stdlinux, you are out of luck.**

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 <loop> is now <loop> ::= **do** <stmt-seq> **while** <cond> **enddo ;** Unlike a while-do loop, a do-while loop is executed at least once. At the end of the loop body, if the condition is true, the loop continues with its next iteration. If the condition is false, the loop terminates.

The semantics of this modified language should be obvious; if some aspects of the semantics are not clear, talk to me. You are supposed to write an interpreter for the *exact language* described in this handout: 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" 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 (i.e., you should be able to access them using the parameters of `main`). 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 variables, the input values in the second file will be integers, separated by spaces and/or tabs and/or "end-of-line". Note that these integers could be negative (e.g., the input stream 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 the token stream. As discussed in class, getting the tokens is typically done on demand: the parser asks the scanner for the current token, or to move to the next token. There are two options for creating the token stream: (1) upon its initialization, the scanner reads from the file the entire program, tokenizes it, and stores all tokens in some list or array, or (2) upon its 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 use (1) or (2), whichever is more convenient for you.

The input program contains a non-empty sequence of ASCII characters; to get them, you should use the C/C++/Java libraries for I/O. The interpreter should exit back to the operating system after processing the entire sequence of input characters (i.e., when end-of-file is reached). If an unexpected end-of-file is encountered (e.g., in the middle of a program), an 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

Input line 1: <*tab*><*tab*>**`program`**<*space*><*eol*>
Input line 2: <*space*>**`int`**<*tab*><*eol*>
Input line 3: `x`<*space*>`,y,`<*tab*>`z;` <*eol*>
Input line 4: **`begin`**<*space*>**`input`**<*space*>`x,y;z:=x;x`<*space*>`:=y;y:=`<*eol*>
Input line 5: `z;`**`output`**<*eol*>
Input line 6: `x,y`<*space*><*space*>`;`<*tab*>**`end`**<*space*><*tab*><*space*><*eof*>

Here <*eol*> represents the ASCII "end of line" representation. In Linux, this is ASCII character 10 (LF, line feed). The end of file is represented by <*eof*>. For this input your scanner should produce the following sequence of tokens:

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

Feel free to add a convenience token SCANNER_ERROR, if necessary. The scanner is case-sensitive. An id is <id> ::= <letter> | <id><letter> | <id><digit>. A keyword takes precedence over an id: e.g., "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 (e.g., flex, jflex) that allow complex text processing. Your implementation of the scanner should use only the library functionality from stdlib.h/string.h (in C/C++) or java.lang.String (in Java), as well as basic library functions for file I/O. 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 issues, ask a question in class.

Syntax Analysis

The parser processes the stream of tokens produced by the scanner, and builds the parse tree representation. Do not make changes to the grammar and then expect the grader to rewrite the test cases.

Parse Tree Representation

You need to implement a ParseTree data type (typically, a class in C++/Java with private fields and public methods, or a set of C functions working on a common private data structure). 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 size 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 assumptions – that is, assume no limits on the number of nodes, ids, and characters in ids.

If you prefer, you may also use the object-oriented approach we discussed in class, with a separate class corresponding to each non-terminal of the grammar. Here again you must make sure that you do not violate the principles of data abstraction – the access to the internal data should be through a "protective wall" of methods (at the very least, fields should be private and not directly accessed).

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 as possible: e.g., instead of

```
  input <space>x, <space>y<space>;
  z<space>:=<space>x<space>;
```

use

```
  input <space>x,y;
  z:=x;
```

Here are examples of pretty printing for some of the productions:

```
program
<space><space>int x,y,z,w;
begin
<space><space>input y,z,w;
<space><space>x:=y+z;
<space><space>if![5<6]then
<space><space><space><space>w:=x*z;
<space><space>else
<space><space><space><space>w:=y*z;
<space><space><space><space>if[x>=y]
<space><space><space><space><space><space>x:=y+z;
<space><space><space><space>endif;
<space><space>endif;
end
```

For the executor, each output integer 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 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 describe the source of the problem. Do not try to have some sophisticated error messages and error handling. 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, your interpreter should not "crash" with a segmentation fault, uncaught exceptions, etc. *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* 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 errors any multiply-declared variables: e.g. "int x,y; int z,x;".

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 tries to execute 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

should 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 significant amount of time on the scanner and the parser. Once they are working correctly, the printer and the executor should be relatively straightforward.
- Pick a small subset of the language (e.g., only a few of the grammar productions) and implement fully-functioning scanner and parser for that subset. Do paranoid testing. Add more grammar productions. Repeat.
- Start early, early, early. The project is too complex to be completed in a few days.

Project Submission

On or before **11:59 pm**, **October 17** you should submit the following:

- One or more files for the interpreter (source code). No .o files, no .class files.
- 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 (1 line) description of each stating what the file contains
  - Instructions for the grader about how to compile and run the project
  - Any special issues to remember during compilation or execution
- A documentation file (also ASCII text). This file should contain at least 50 non-empty lines, and should include at least the following:
  - 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 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).
  - If you borrowed ideas or anything else (e.g. code) from anywhere, describe briefly.

Submit your project using the following command at the stdlinux UNIX shell prompt

```
submit c3341ac lab1 file1 file2 …
```

Submit only the files described above: no x.o, x.class, x.doc, etc.

If the time stamp on your electronic submission is **12:00 am on October 18 or later**, you will receive 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, this will override any previous submissions and only **the latest** submission will be considered – **resubmit at your own risk**.

If the grader has problems compiling or executing your program, she/he will e-mail you; you **must** respond within 48 hours to resolve the problem. Please check often both your xyz.567@osu.edu and xyz@cse.ohio-state.edu accounts after submitting the project (for about a week or so) in case the grader needs to get in touch with you.

Grading

The grader will build and run your project as described in your README file, using an extensive set of test cases. The grader will then read the documentation and parts of your code, and will assign a grade. The project is worth 100 points. Correct functioning of the interpreter is worth 65 points, with partial credit in case your interpreter works for some cases but not all, or parses but does not execute, etc.. The handling of error conditions is worth 20 points. The grader will use an extensive set of "bad" inputs to test the error-handling of your interpreter. The implementation style and the documentation are worth 15 points. In particular, make sure you observe data abstraction principles, and your code is reasonably readable, commented, and organized.

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 for 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). I strongly recommend that you check this URL. If you have any questions about university policies or what constitutes academic misconduct in this course, please contact me immediately.