CSE 3341 Project 3 - CORE Interpreter

Overview

The goal of this project is to build an interpreter for the Core language discussed in class. This project should be completed using Java or Python.

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.

How you interpreter should represent variables/memory, and 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. Building from the project 2 parser, you need to write an executor 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 three main components: a lexical analyzer (a.k.a scanner from project 1), a syntax analyzer (a.k.a. parser from project 2), and an executor. The executor should be written using the recursive descent approach discussed in class, similar to how the parser in project 2 worked.

Main

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

1. Initializes the scanner.
2. Parses the input program.
3. Execute the input program.

You can add extra steps here if you like, for example you may choose to add some extra checks or modify the parse tree after parsing and before executing. You do not need to print the program or run your semantic checks from project 2.

The Executor

The goal of this project is to use recursive descent to walk over the parse tree built by your parser, and "execute" the Core program by performing the actions described. Essentially this means that for each "parse" function, you will need to make an "execute" function, which will execute its children and perform any action needed on the result of that execution.

For example, if you followed my suggestion to have a class for each nonterminal in the grammar, you will want to add to you Program class an execute function that is something

like this:

```
class Program {
    DeclSeq ds;
    StmtSeq ss;

    void parse() {
        // Probably nothing needs to change here
    }

    void execute() {
        if (ds != null) ds.execute();
        ss.execute();
    }
}
```

Many of the execute functions will be this simple. Others will require something more complicated and you may decide to pass in values or have values returned. For example, for Expr, Term, Factor, and similar things it probably makes sense to have a return value so the execute function can look something like this:

```
class Expr {
    Term t;
    Expr e;
    int option;

    void parse() {
        // Probably nothing needs to change here
        // Assume I set ``option'' here based on which production:
        //   option=0 if just a term
        //   option=1 if addition
        //   option=2 if subtraction
    }

    int execute() {
        int value = t.execute();
        if (option==1) {
            value = value + e.execute();
        } else if (option==2) {
            value = value - e.execute();
        }
        return value;
    }
}
```

Memory and Variables

Your interpreter should maintain 3 "regions" of memory: Static, Stack, and Heap.

1. Global variables (any int or ref variables created in the DeclSeq) should be allocated in the static region. I suggest creating a map from string to integer.

2. Local variables (any non-global variable) should be allocated in the stack region. I suggest using a stack of maps from string to integer. You can then push a new map onto the stack each time a new scope is entered, and pop a map from the stack each time a scope is exited (i.e. at the start and end of an if statement).

3. The heap can be represented by an array or list of integers. We will want the ref variables to behave like references; we will simulate this by having the ref variables store an index into the heap.

I suggest putting these data structures in their own class, along with any helper functions you create to interact with them. With memory structured in this way, we have the following semantics for how variables are used:

1. When an int variable is declared, it has initial value 0.

2. When a ref variable is declared, it is initially a null reference.

3. When an int variable is used in a factor, we need to fetch the value stored for that variable in the stack or static region.

4. When a ref variable is used in a factor, we need to fetch the value stored for that variable in the stack or static region and then use that as an index to get a value from the heap.

5. When an int variable is used as the left hand side of an assignment statement, we need to change the value of the variable in the stack or the static region.

6. When a ref variable appears on the left hand side of an assignment:

   (a) For "id = < *expr* >" type assignment, we need to get the value of the variable from the stack or static region, then use that as an index into the heap and store the value of the expression there.

   (b) For "id = new class" type assignment, we need to create a new position in the heap (you can just keep adding to the end of the array or list) and change the value of the variable in the stack or heap to the index of this new position.

   (c) For "id = share id" type assignment, we need to copy the stack or static region value of the variable on the right hand side to the stack or static region value of the variable on the left hand side.

Input To Your Interpreter, The Input Function in Core

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 will be a .code file that contains the program to be executed.

The second file will be a .data file that contains integer constants separated by spaces. Creating another instance of your scanner should make getting these constants easy.

During execution, each Core **input()** function will read the next data value from the second file each time it is executed, to simulate user input.

Output From Your Interpreter, The Output Statement in Core

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

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

For the executor, each Core **output** statement should print on a new line the integer value of the expression, without any spaces/tabs before or after it. Other than that, the executor should only have output if there is an error. The output for error cases is described below.

Invalid Input

We will not be rechecking the error handling of your scanner and parser, you can focus on just catching runtime errors in your executor.

There are just two kinds of runtime errors possible in the current version of Core:

The first kind of error is with the input function. If an input function is executing but all values in the .data file have already been used then your executor should print a meaningful error message and halt execution.

The second kind of error is with the ref variables. If we attempt an "id = input()" or "id = < *expr* >" type assignment to a ref variable that currently has a "null" value, that should result in an error.

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 there are three files: a .code file, a .data file, and a .expected file.

I will provide a "tester.sh" script that works similar to how the script from project 2 worked, and it will use the diff command to compare your output to the expected output. Your output should exactly match what is in the .expected files.

Suggestions/Notes

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

- Just like I suggested for the parser, take the executor in stages. Implement just enough to execute a very simple program, then test, then implement more, ect.
- Before starting, spend some time thinking about these things:
  - How to to keep track of the value of each variable (symbol table)?
  - How to handle the "hiding" of variables with name conflicts as we move into new scopes?
- While constants in Core are integers 0-255, variables in Core can store positive and negative integer values. You do not need to worry about the range or overflow here, just represent them using the built in integer representation for Java and Python.
- 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.
- Remember that the grammar enforces right-associativity. In Core we have $1-2-3 = 2$, 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, in particular how you are handling variables w.r.t. tracking their values and hiding variables when entering a new scope
  - 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 3.

If the time stamp on your submission is 12:00 am on June 18th 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 75 points. The handling of error conditions is worth 10 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.

Please note this is a language like C or Java where whitespaces have no meaning, and whitespace can be inserted between keywords, identifiers, constants, and specials to accommodate programmer style. This grammar does not include formal rules about whitespace because that would add immense clutter.

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

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

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

<decl> ::= <decl-int> | <decl-ref>

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

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

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

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

<assign> ::= **id = input ( ) ;** | **id =** <expr> **;** | **id = new class;** | **id = share id ;**

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

<if> ::= **if** <cond> **then {** <stmt-seq> **}**

| **if** <cond> **then {** <stmt-seq> **} else {** <stmt-seq> **}**

<loop> ::= **while** <cond> **{** <stmt-seq> **}**

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

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

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

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

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

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

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