CSE 3341 Project 4 - Function Calls

Overview

The goal of this project is to modify the Core interpreter from Project 3 to now handle function definitions and function calls.

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.

You should not need to modify the scanner for this project, but you are allowed to if you want to. You will need to modify the parser and executor functions to accommodate functions. I also recommend modifying the printer functions so you can verify your parser functions are working properly, but this is not required.

Functions

On the last page is the modified Core grammar to allow functions. Here is an example of a Core program that defines and calls a function:

```
program
        int x, y, z;
        add(a, b, c) begin a=b+c; b=0; c=0; endfunc;
begin
        x=1;
        y=2;
        z=3;
        begin add(x,y,z);
        output x;
        output y;
        output z;
end
```

For simplicity our parameter passing will be done with **call by copy return** - the values of the actual arguments are copied to the formal parameters, the function executes, and then the values of the formal parameters are copied back to the actual parameters.

So what should happen here is that the values of x, y, z are copied to a, b, c, then the statements a=b+c; b=0; c=0; are evaluated. When the function is done, the values of a, b, c are copied back to x, y, z. In this example, after the function call we should have x=5, y=0, z=0.

Your parser should check that ids in the <id-list> part of the function definition are distinct from each other, but the ids here do not need to be distinct from ids in other functions or from the int variables declared. For example, this is a valid declaration sequence:

```
int x,y,z;
x(a,b) begin a=b; endfunc;
y(x,y,z) begin x=y*z; endfunc;
```

Your interpreter should support recursion. This will require implementing a call stack.

Input to your interpreter

The input to the interpreter will be same as the input from Project 3:

Input 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.

Parse Tree Representation

You should create new nodes for the new parts of the grammar, which are highlighted in blue on the last page.

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.

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

With the addition of functions to the language, after the parse tree is constructed you should make sure that every function declared has a unique name, two functions with the same name should result in an error. You should also make sure that each function call has a valid target, i.e. there is a function defined in the declaration sequence that matches the name and number of arguments in the call.

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. Like the previous projects, I will provide a tester.sh script to help automate your testing.

Project Submission

On or before 11:59 pm November 9th, 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 the call stack is implemented.
  - 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 4.

If the time stamp on your submission is 12:00 am on November 10th 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> | <decl-func> | <decl-func><decl-seq>

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

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

<decl-func> ::= **id (** <id-list> **) begin** <stmt-seq> **endfunc ;**

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

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

<func> ::= **begin id (** <id-list> **) ;**

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