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 will need to modify the parser and executor functions to accommodate function calls.

You should not need to modify the scanner for this project, but you are allowed to so if you want. You also do not need to modify the existing printer functions or add printer functions for the new parts of the grammar, but it might help you check your work and troubleshoot bugs if you do implement print functions for the new classes.

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

For simplicity functions will only accept ref variables as actual parameters, and our parameter passing will be done with **call by sharing** - the reference value of the actual parameters are copied to the formal parameters. In this example, after the function call we should have output 5, then 0, then 0.

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.

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 a syntax error. You should implement checks to ensure the modified grammar is being followed.

For the executor, each Core output statement should produce an integer printed on a new line, without any spaces/tabs before or after it. Other than that, the executor functions should only have output if there is an error.

Invalid Input - Syntax Checks

For this project you will need to modify some existing parse functions, and add new classes and new parse functions. Your parse functions should ensure the syntax rules of the modified grammar are being followed.

Invalid Input - Semantic Checks

With the addition of functions to the language you will need to perform some additional semantic checks. 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 (we will not allow overloading).

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. You **do not** need to implement checks to verify the arguments are all ref variables, we will consider using int variables as arguments as undefined by the language.

Your parser should check that the formal parameters of a function are distinct from each other, but the formal parameters of one function do not need to be distinct from the formal parameters of another function, or from any global or local variables. For example, this is a valid declaration sequence:

```
int x,y,z;
x(ref a,b) begin
	int a, b;
	b=1;
	a=b;
endfunc
y(ref x,y,z) begin
	x=y*z;
endfunc
```

Global variables can be accessed within the body of a function, and functions can be called from another function. To help keep your checks simple, you may assume functions and global variables will always be declared before being used.

Implementation Suggestions

Here are my suggestions for how to implement the project:

1. Focus first on modifying/creating parse functions, and add new print functions to verify your modified parser. There is no point in trying to do the execution unless you are confident in your parse tree!

2. You will need a stack of "frames" to allow recursion. A frame can just be whatever data structure you used to represent the "Stack Space" from project 3. Create a stack of these frames, and always "peek" at the frame on the top of the stack.

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 Novvember 5th, 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 6th 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.

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 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> | <func-decl> | <func-decl><decl-seq>

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

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

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

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

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

<func-decl> ::= id (ref <formals>) begin <stmt-seq> endfunc

<formals> ::= id | id , <formals>

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

<func-call> ::= begin id (<formals>) ;

<assign> ::= **id =** <expr> **;** | **id = new ;** | **id = ref id ;**

<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> **)**