CSE 3341 Project 4 - Procedure Calls

Overview

The goal of this project is to modify the Core interpreter from Project 3 to now handle procedure declarations and procedure 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.

Procedures

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

```
procedure test is
        procedure add(object a, b) is
                integer x;
                x=a[a]+b[a];
                print(x);
        end
        object x;
        object y;
begin
        x = new object(a, 1);
        y = new object(a, 1);
        x[a] = 1;
        y[a] = 2;
        begin add(x,y);
        print(x[a]);
        print(y[a]);
end
```

For simplicity you may assume procedures will only accept object variables as actual parameters, and our parameter passing will be done with **call by sharing** - **x** and **a** point to the same object, **y** and **b** point to the same object. In this example, as output we should see 3, then 1, then 2.

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, a .code and a .data file

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 syntax checks to ensure the modified grammar is being followed.

For the executor, each Core print 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 procedures to the language you will need to perform some semantic checks. You **DO NOT** need to include the semantic checks from project 2, you only need to handle the new ones described here. You can perform the semantic checks during parsing, during execution, or separately.

Here are the semantic rules you should be checking:

1. Every procedure should have a unique name (no overloading).
2. Each procedure call has a valid target.
3. Your parser should check that the formal parameters of a procedure are distinct from each other. You **do not** need to implement checks to verify the arguments are all distinct or are all object variables, we will consider using integer variables as arguments as undefined by the language.

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 even thinking about the execution unless you are confident in your parse tree!

2. You will need a stack of "frames" to allow recursion. A frame should be the same as your local memory from project 3. You then need a stack of frames, so if you used my suggestions for project 3 then for project 4 you will have a stack of stack of maps!

3. Implement the semantic checks last.

4. During execution you will need a way to "jump" across the tree, or in other words when I am in the StmtSeq and I want to call a procedure named "Foo", I need some way of finding the part of the tree where "Foo" is defined in the DeclSeq.

5. The hardest part will be implementing the execute function for the procedure call. To execute a procedure call, you need to:

   (a) Create a new frame

   (b) Create the formal parameters in this frame and copy over the values of the arguments

   (c) Push the frame to the top of the stack

   (d) Execute the body (StmtSeq) of the procedure

   (e) Pop the frame off 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 June 20th, 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 June 21st 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 formal rules about whitespace because that would add immense clutter.

<procedure> ::= **procedure id is** <decl-seq> **begin** <stmt-seq> **end**

| **procedure id is begin** <stmt-seq> **end**

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

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

<decl> ::= <decl-integer> | <decl-obj>

<decl-integer> ::= **integer id ;**

<decl-obj> ::= **object id ;**

<function> ::= **procedure ID (object** <parameters> **) is** <stmt-seq> **end**

<parameters> ::= **ID** | **ID ,** <parameters>

<stmt> ::= <assign> | <if> | <loop> | <print> | <read> | <decl> | <call>

<call> ::= **begin ID (** <parameters> **) ;**

<assign> ::= **id =** <expr> **;** | **id [string] =** <expr> **;** | **id = new object(string,** <expr> **);** | **id : id ;**

<print> ::= **print (** <expr> **) ;**

<read> ::= **read (id) ;**

<if> ::= **if** <cond> **then** <stmt-seq> **end**

| **if** <cond> **then** <stmt-seq> **else** <stmt-seq> **end**

<loop> ::= **for (id =** <expr> **;** <cond> **;** <expr> **) do** <stmt-seq> **end**

<cond> ::= <cmpr> | **not** <cond> | **[** <cond> **]** | <cmpr> **or** <cond> | <cmpr> **and** <cond>

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

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

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

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