CSE 3341 Project 5 - Garbage Collection

Overview

The goal of this project is to modify the Core interpreter from Project 4 to now handle references and garbage collection.

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.

Reference Variables

On the last page is the modified Core grammar to allow declaring reference variables. Examples of new parts of the language and their meaning are described here:

1. define x;
   This is a new statement that declares a reference variable x but does not allocate anything on the heap (x is a null reference).
   Note that reference variables can not be declared in the declaration sequence (no global reference variables).

2. x = new 10;
   Using the new keyword in this way should put the value on the heap and x is a reference to that value.

3. 2+x
   When reference variable x is used in an expression, we use the value x is a reference to. Given the previous examples, this expression should evaluate to 12.
   If x is a null reference, the behavior is undefined (you do not need to implement error checks).

4. x = define y;
   If y was declared using "define y;" then we want to copy the reference from y to x. If y is a null reference, x will also be a null reference after this.
   If y was declared using "int y;" then the behavior is undefined (you do not need to implement error checks).

5. begin A(x);
   When reference variable x is used in a function call, the corresponding formal parameter should also be created as a reference variable. Reference variables should be passed using "call by sharing return" - a new parameter passing mode in which the reference is copied from actual parameter to formal parameter before the function executes, and then from formal parameter back to actual parameter after the function finishes (the effect of this parameter passing will be very similar to call by reference).

The Heap and the Garbage Collector

You should implement a “heap” for storing values. I suggest a list. Please make sure you describe your heap in your readme file.

You should implement a reference counting garbage collector for your heap.

Input to your interpreter

The input to the interpreter will be same as the input from Project 4 (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/semantic check functions should only produce output in the case of a syntax or semantic 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.

The garbage collector should output the current number of values on the heap each time that number changes. Please follow this format: “gc:n” where n is the number of values.

For example, consider the following program:

```
program
    add(a, b) begin
        a = new a+b;
        define n;
        b = define n;
    endfunc
begin
    define x;
    x = new 1;
    define y;
    y = new 1+x;
    begin add(x, y);
    output x;
end
```

This program should print these values at the following times:

1. gc:1 when x = new 1; executes
2. gc:2 when y = new 1+x; executes

3. gc:3 when a = new a+b; executes
4. gc:2 then gc:1 when the call to A returns
5. 3 when output x; executes
6. gc:0 when program ends and x goes out of scope

Invalid Input

Make sure you implement checks to verify the grammar is being followed.

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.

Implementation Suggestions

Here are some suggestions for how to implement this project:

1. Represent your heap as a list of integers.
2. Create another list of integers to store your reference counts.
3. Reference variables can be created just like the int variables, and they can store be the index of the value in the heap they reference.
4. You will need some way of distinguishing int variables from reference variables. ~~One way of handling this would be to add a prefix to the variables during parsing or semantic checks. For example, if 'x' was declared as an int rename it to '0x' and if 'x' was declared as a reference rename it to '1x'.~~
   The previous suggestion would not work due to how I have define function calls with reference variable to work. What I have done instead in my implementation is create during execution a map from string to stack of integer. Each time a variable is declared a new value is pushed onto the corresponding stack, I use 0 of it is an int and 1 if it is a reference. When a variable goes out of scope, I pop a value off the corresponding stack.

Project Submission

On or before 11:59 pm April 2nd, 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 5.

If the time stamp on your submission is 12:00 am on April 3rd 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 so 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> ::= **int** <id-list> **;**

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

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

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

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

<new-decl> ::= define **id**;

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

<assign> ::= **id =** <expr> **;** | **id** = new <expr> ; | **id** = define **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> **)**