CSE 3341 Project 5 - Garbage Collection

Overview

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

The Heap and the Garbage Collector

You should implement reference counting for the values stored on the heap.

My suggestion for this is to create a "reference count list" that is a duplicate of your heap data structure, and use this to keep track of the reference counts. Each time the heap grows, also add additional positions to the reference count list.

As ref variables (our references) are manipulated or go out of scope, you will be incrementing and decrementing the reference counts. Once a reference count reaches 0, that position in the heap can be garbage collected - **you do not need to actually perform this garbage collection by removing values from the heap**. Instead the garbage collector will produce some output to so we can verify it is identifying the correct points to perform a garbage collection.

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.

Just like in the last project, each Core output statement should produce an integer printed on a new line, without any spaces/tabs before or after it.

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

For example, consider the following program:

```
program {
        add(ref a, b) {
                a = a+b;
                ref n;
                n = new instance;
                n = 10;
        }
begin {
        ref x;
        x = new instance;
        x = 5;
        ref y;
        y = new instance;
        y = 7;
        begin add(x, y);
        output(x);
}}
```

During execution, this program should produce the following output (in bold):

1. **gc:1** when x = new instance; executes
2. **gc:2** when y = new instance; executes
3. **gc:3** when n = new instance; executes
4. **gc:2** when the call to add returns (n has gone out of scope, we've lost the last reference to where we put that 10)
5. **12** when output(x); executes
6. **gc:1** then **gc:0** when program ends and x, y go out of scope

Invalid Input

There are no new errors we need to look for here.

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 18th, 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 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 November 19th 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 85 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> **} }** | **program { begin {** <stmt-seq> **} }**

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

<func-decl> ::= id (ref <formals>) { <stmt-seq> }

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

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

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

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