Programming Languages (CS 421/521), Spring 2019

Assignment 04 (100 points)

Due 5:00 PM, Friday, 26 April 2019

Consider the simple programming language given by the following EBNF grammar, where *Program* is the starting symbol:

$$
\begin{array}{rcl}
\mathit{Program} & \longrightarrow & \mathit{Declaration}^\star\ \mathit{Method}^\star\ \mathit{Main} \\
\mathit{Method} & \longrightarrow & \mathit{Void} \mid \mathit{NonVoid} \\
\mathit{Void} & \longrightarrow & \texttt{void}\ \mathit{Name}\texttt{(}\ \mathit{Parameters}\ \texttt{)}\ \texttt{\{}\ \mathit{Statement}^\star\ \texttt{\}} \\
\mathit{NonVoid} & \longrightarrow & \texttt{int}\ \mathit{Name}\texttt{(}\ \mathit{Parameters}\ \texttt{)}\ \texttt{\{}\ \mathit{Statement}^\star\ \mathit{Return}\texttt{\}} \\
\mathit{Name} & \longrightarrow & \mathit{Letter}\ \mathit{Letter}^\star \\
\mathit{Parameters} & \longrightarrow & \epsilon \mid \texttt{int}\ \mathit{Variable}\ (\texttt{,}\ \texttt{int}\ \mathit{Variable})^\star \\
\mathit{Return} & \longrightarrow & \texttt{return}\ \mathit{Expression}\texttt{;} \\
\mathit{Main} & \longrightarrow & \texttt{int main() \{}\ \mathit{Statement}^\star\ \texttt{\}} \\
\mathit{Statement} & \longrightarrow & \mathit{Declaration} \mid \mathit{Assignment} \mid \mathit{Call} \\
\mathit{Declaration} & \longrightarrow & \texttt{int}\ \mathit{Variable}\texttt{;} \mid \texttt{int}\ \mathit{Assignment}\texttt{;} \\
\mathit{Assignment} & \longrightarrow & \mathit{Variable}\ \texttt{=}\ \mathit{Expression}\texttt{;} \\
\mathit{Expression} & \longrightarrow & \mathit{Int} \mid \mathit{Variable} \mid \mathit{BinaryOperation} \mid \mathit{Call} \\
\mathit{Call} & \longrightarrow & \mathit{Name}\texttt{(}\ \mathit{Arguments}\ \texttt{);} \\
\mathit{Arguments} & \longrightarrow & \epsilon \mid \mathit{Argument}\ (\texttt{,}\ \mathit{Argument})^\star \\
\mathit{Argument} & \longrightarrow & \mathit{Variable} \mid \mathit{Int} \\
\mathit{Variable} & \longrightarrow & \mathit{Letter}\ \mathit{Letter}^\star \\
\mathit{Letter} & \longrightarrow & \texttt{a} \mid \texttt{b} \mid \ldots \mid \texttt{z} \\
\mathit{Int} & \longrightarrow & \texttt{0} \mid \mathit{LeadDigit}\ \mathit{Digit}^\star \\
\mathit{LeadDigit} & \longrightarrow & \texttt{1} \mid \texttt{2} \mid \ldots \mid \texttt{9} \\
\mathit{Digit} & \longrightarrow & \texttt{0} \mid \texttt{1} \mid \ldots \mid \texttt{9} \\
\mathit{BinaryOperation} & \longrightarrow & \mathit{Operator}\ \mathit{Expression}\ \mathit{Expression} \\
\mathit{Operator} & \longrightarrow & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/}
\end{array}
$$

That is, programs in this language consist of a `main()` method preceded by 0 or more global variable declarations and/or assignments, and and 0 or more other methods. The only basic type is `int`, and all methods return `void` or `int`. Each method has 0 or more inputs. The body of any method consists of variable declarations, assignments, applications of (prefix form) binary arithmetic and method calls, where `void` methods can be called as single statements by themselves, while `int` methods can be called in assignment contexts, where returned values are assigned to some variable.

The language will use pass-by-value semantics for method calls, and will allow local variables to mask global ones. (Local variables can not mask method parameters.)

A sample program is as follows (we will assume that extra lines of white space and usual indenting is allowed); after this program has executed, the value of `j` in `main()` will be `200`.

```
int a = 1;
int b;

int helper(int x, int y) {
   int i = + x y;
   b = i;
   int j = - i a;
   return j;
}

int main() {
   int i = 100;
   int j = 101;
   j = helper(i, j);
}
```

Your job is to write a **semantic simulator** that deals with methods, as described in lectures 23–26. That is, your program will do the following:

1. It will take the name of a text-file containing a valid program like the above as a command line argument.
2. It will read in the text-file and parse it, printing out semantic information as follows:
   (a) At each stage, it will print out information corresponding to the semantic triple $\sigma = \langle \gamma, \mu, a \rangle$, where γ is the set of $\langle identifier, address \rangle$ pairs, μ is the memory set of active $\langle address, value \rangle$ pairs, and a is the stack pointer.
   (b) It will begin by printing out the information for the initial state σ_0, followed by the state after any global variables are defined/assigned.
   (c) Each time a method f is *called*, it will print out the state σ_f at the *start* of that method.
   (d) Each time a method f *returns*, it will print out the state σ_f at the *end* of that method.

We will assume that all programs input are valid with the usual rules. That is, all variables used in a method are properly declared, every call to a method uses the right number of input parameters, etc. We will also assume that our machine memory locations are all *unused* unless otherwise specified, and we do not need to list the unused locations.

Thus, for the example program, your code would print out information as follows (formatting needs to be as close to the example given as possible to make grading simpler):

```
sigma_0:
  gamma: {}
  mu: {}
  a: 0

sigma_global:
  gamma: {<a, 0>, <b, 1>}
  mu: {<0, 1>, <1, undef>}
  a = 2

sigma_main_in:
  gamma: {<a, 0>, <b, 1>, <i, 2>, <j, 3>}
  mu: {<0, 1>, <1, undef>, <2, undef>, <3, undef>}
  a = 4

sigma_helper_in:
  gamma: {<a, 0>, <b, 1>, <x, 4>, <y, 5>, <i, 6>, <j, 7>, <helper, 8>}
  mu: {<0, 1>, <1, undef>, <2, 100>, <3, 101>, <4, 100>, <5, 101>,
       <6, undef>, <7, undef>, <8, undef>}
  a = 9

sigma_helper_out:
  gamma: {<a, 0>, <b, 1>, <x, 4>, <y, 5>, <i, 6>, <j, 7>, <helper, 8>}
  mu: {<0, 1>, <1, 201>, <2, 100>, <3, 101>, <4, 100>, <5, 101>,
       <6, 201>, <7, 200>, <8, 200>}
  a = 9

sigma_main_out:
  gamma: {<a, 0>, <b, 1>, <i, 2>, <j, 3>}
  mu: {<0, 1>, <1, 201>, <2, 100>, <3, 200>}
  a = 4
```

The assignment comes with some sample program files. You can write your own examples to help in writing and testing. When you are done, submit an archive consisting of your source-files and a `README` with instructions as to how to compile/execute your program.