Homework: Syntax and Semantics of Programming Language with Basic References

Due-date: Apr 17 at 11:59pm
Submit online on Canvas

Homework must be individual's original work. Collaborations and of any form with any students or other faculty members are not allowed. If you have any questions and/or concerns, post them on Piazza and/or ask 342 instructor or TAs.

Learning Outcomes

- Knowledge and application of Functional Programming
- Ability to understand grammar specification
- Ability to design software following requirement specifications (operational semantics for references)

Questions

Consider the grammar G of a language $\mathcal{L}$, where $G = (\Sigma, V, S, P)$ such that

- Σ is a set of terminals: anything that does not appear on the left-side of the product rules P presented below
- V is the set of non-terminals appearing in the left-side of the production rules P presented below

```
Program      -> (SSeq)
SSeq         -> Statement | Statement SSeq
Statement    -> Decl | Assign | If | While | FunDecl | FunCall | PStmt
Decl         -> (decl Var)
Assign       -> (assign Var ArithExpr)
If           -> (if CondExpr (SSeq))
While        -> (while CondExpr (SSeq))
FunDecl      -> (fundecl (FName ParamList) (SSeq))
FunCall      -> (call (FName ArgList) 1)
ParamList    -> () | (Params)
Params       -> Var | Var Params
ArgList      -> () | (Args)
Args         -> ArithExpr | ArithExpr Args
FName        -> symbol

PStmt        -> (deref Var ArithExpr) | (ref Var ArithExpr)
               | (free ArithExpr) | (wref ArithExpr ArithExpr)


ArithExpr    -> Number | Var | (Op ArithExpr ArithExpr) | Anonf
Op           -> + | -  | * | /
CondExpr     -> BCond | (or CondExpr CondExpr) |
                           (and CondExpr CondExpr) | (not CondExpr)
BCond           -> (gt ArithExpr ArithExpr) | (lt ArithExpr ArithExpr) |
                   (eq ArithExpr ArithExpr)
Var             -> symbol
Anonf           -> (anonf ((Params) ArithExpr) (Args))
```

- $S =$ `Program`

Objective. Write a function `sem` which has three formal parameters:

1. a syntactically correct program as per the above grammar
2. an environment
3. a heap

and computes the semantics of the program thus producing a list containing two elements, where the first element is an environment and the second element is heap.

(Recall that as a program is a sequence of statements, the semantics of each statement is also based on some environment and heap, and the output of semantic evaluation of a statement is a new environment and heap. For a consecutive pair of statements s_i s_j, the semantic evaluation of s_i produces the environment and the heap for the semantic evaluation of s_j.)

The input environment is a list of variable-values and function-definitions as per the grammar specified in the homework 5. The input heap is a list, where each element itself is a list containing the address (some number) of a location and associated value (value can be a number or the symbol `free`). Address associated to each location are unique.

The output environment has the same format as the input environment: it contains the variable-value pairs and function-name-definition pairs that are accessible in the outermost scope of the program (block contexts definitions follow from previous assignments). The output heap has the same format as the input heap: it contains the updates to the heap as a result of semantic evaluation of the program. If the semantic evaluation leads to some exception, then the output heap is a list containing only a symbol indicating the type of exception (which is `ooma`, `oom` or `fma`; see below for description).

The newly introduced statement constructs for this assignment are captured by the non-terminal `PStmt`. The semantic definitions for all other expressions and statements, and related assumptions follow from the previous assignments.

The semantic rules for the newly introduced statement constructs are as follows:

1. `(deref X ArithExpr)`: Compute the semantics of `ArithExpr`, which will produce a number L. Find the value v associated to location L in the heap. Update the value of (appropriate) variable `X` in the environment with v.

   If location L does not exist in the heap, then the exception is `ooma` (out-of-memory-access) and the resultant heap becomes a list containing the symbol `ooma`. If the location L is associated to a symbol `free`, then the exception is `fma` (free-memory-access) and the resultant heap becomes a list containing the symbol `fma`.

2. `(ref X ArithExpr)`: Compute the semantics of `ArithExpr`, which will produce a number v. Find the first free heap area; let the location for that free area is L (it is associated to symbol `free`). Update the value at location L with v. Update the value of (appropriate) variable `X` in the environment with L.

   If free heap area cannot be found, then the exception is `oom` (out-of-memory), and the resultant heap becomes a list containing the symbol `oom`.

3. `(wref ArithExpr1 ArithExpr2)`: Compute the semantics of `ArithExpr1` to produce a number L and compute the semantics of `ArithExpr2` to produce a number v. Update the heap location L by the value v.

   If location L does not exist in the heap, then the exception is `ooma` (out-of-memory-access) and the resultant heap becomes a list containing the symbol `ooma`. If the location L is associated to a symbol `free`, then the exception is `fma` (free-memory-access) and the resultant heap becomes a list containing the symbol `fma`.

4. `(free ArithExpr)`: Compute the semantics of `ArithExpr` to produce a number L. Update the heap such that the value associated to the location L becomes the symbol `free`.

   If location L does not exist in the heap, then the exception is `ooma` (out-of-memory-access) and the resultant heap becomes a list containing the symbol `ooma`.

Programming Rules

- You are required to submit one file hw6-⟨net-id⟩.rkt[1]. The file must start with the following.

  ```
  #lang racket
  (require "program.rkt")
  (provide (all-defined-out))
  ```

  In the above, the program.rkt will be used as an input file for our test programs.

- You are **only allowed** to use functions, if-then-else, cond, basic list operations (including append and sort), operations on numbers. No imperative-style constructs, such as begin-end or explicitly variable assignments, such as get/set are allowed. If you do not follow the guidelines, your submission will not be graded. If you are in doubt that you are using some construct that may violate rules, please contact instructor/TA (post on Piazza).

- You are expected to test your code extensively. If your implementation fails any assessment test, then points will be deducted. Almost correct is equivalent to incorrect solution for which partial credits, if any, will depend only on the number of assessment tests it successfully passes.

- We will assess correctness and (to a lesser extent) efficiency of implementation. Avoid repeated computations of the same expressions/statements.

[1]Your netid is your email-id and please remove the angle brackets, they are there to delimit the variable net-id.

Here are few example programs to get you started with testing.

```
(define p0
  '(
    (decl x)
    (decl y)
    (ref x 10)
    (deref y x)
   )
)
; > (sem p0 '() '((1 free) (2 free)))
; '(((y 10) (x 1))
;   ((1 10) (2 free)))

; > (sem p0 '() '((1 20) (2 free)))
; '(((y 10) (x 2))
;   ((1 20) (2 10)))

; > (sem p0 '() '((1 20) (2 40)))
; '(((y 0) (x 0))
;   (oom))


(define p1
  '(
    (decl x)
    (decl y)
    (ref x 10)
    (wref x 30)
    (deref y x)
    (free x)
   )
)
; > (sem p1 '() '((1 free) (2 free)))
; '(((y 30) (x 1)) ((1 free) (2 free)))


(define p5
  '(
     (decl x)
     (assign x 0)
     (free x)
   )
)
; > (sem p5 '() '((1 free)))
;   '(((x 0)) (ooma))

(define p6
  '(
     (decl x)
     (deref x 1)
   )
)
; > (sem p6 '() '((1 free)))
; '(((x 0)) (fma))


(define p2
  '(
     (fundecl (swap (x y)) (
                               (decl temp1)
                               (decl temp2)
                               (deref temp1 x)
                               (deref temp2 y)
```

```
                                (wref x temp2)
                                (wref y temp1)
                              )
      )
      (decl a)
      (decl b)
      (assign a 1)
      (assign b 2)
      (call (swap (a b)) 1)
    )
)
; > (sem p2 '() '((1 20) (2 500)))
;  '(((b 2) (a 1)
;     ((swap (x y))
;                    ((decl temp1)
;                     (decl temp2)
;                     (deref temp1 x)
;                     (deref temp2 y)
;                     (wref x temp2)
;                     (wref y temp1)))
;     )
;    ((1 500) (2 20)))
;
```

Guidelines

1. **Please make sure your submission does not have syntax errors.**
2. **Please remove the trace-directives.**
3. **Please remove any test-code.**
4. **Please include `(provide (all-defined-out))`.**
5. Review and learn about operational semantics. It is important to understand how operational semantics is represented and implemented. Unless you write some on your own, this homework (and subsequent ones) is likely to be difficult.
6. In class, we have developed a language and discussed the implementation of operational semantics of that language. Make sure you understand that before proceeding.
7. Learn how to use Racket. If you have not written some definitions in Racket and have not reviewed the solutions to any/some exercises, then it is likely to be difficult to complete this assignment.
8. Starting two days before the deadline to do the above is likely to make it impossible for you to complete this assignment.
9. As always, map out the functions you need to write, write comments that include the specification of the functions you are writing, test each function extensively before using it in another function.

Directives for completing part of homework 5 as needed in this assignment. If you have started homework 5 from the solution posted for homework 4, the primary changes for implementation of homework 5 as needed for this assignment are as follows (in this assignment, we are only using dynamic scoping - see the grammar):

1. Update findvalue racket-function such that if it cannot find the (integer) valuation of a variable in the environment then that implies the variable is actually a function name, which, in turn, implies that the valuation of the variable is the variable itself.

2. Add the two constructs for functions in semstmt racket-function (one for function declaration and another for function invocation).

3. Write a racket-function that searches the environment for definition and formal parameters for a called function (make sure function with correct number of formal parameters is identified). In this search, if the name of the function matches with the name of a variable, then you will have to recursively continue the search using the "valuation" of the variable.

   For instance, if the environment is of the form: `(... (x f) ...)`, where `x` is assigned to a function name `f` and a call statement `(call ((x ParamList)) 1)` is being interpreted, then your search for `x` will find the mapping `(x f)`. Therefore, you will have to recursively look in the rest of the environment for the function definition using the valuation of `x`, which is `f`.

4. Write a racket-function that associates formal parameters of a called function to valuations of actual arguments (i.e., evaluate the semantics of arguments) of the called function to form an environment in which the semantics of the called function's definition is computed.

   (It may be easier to write a racket-function that computes the valuations of list of arith expressions for computing the semantics of list of actual arguments.)

5. Technique used in making sure environment captures the block context for if-then and while-do in homework 4 solution can be used for block context of functions as well. In particular, as we are only interested in dynamic scoping in homework assignment 6, the function calls can be interpreted simply as inlining the code for function definition in a new block-context.