Homework: Syntax and Semantics of Programming Language with Functions

Due-date: Apr 3 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 higher-order functions)

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

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 takes as input a syntactically correct program as per the above grammar, an environment and computes the semantics, which is another environment. *The output environment contains the variable-value pairs and function-name-definition pairs that are accessible in the outermost scope of the program.*

We will build on the assignment HW4, where you developed the semantics of the programming language with variables, conditional, iteration statements. The solution for that assignment is posted under HW4 (along with a lecture explaining how the solution is developed). You are welcome to develop the solution for HW5 from that solution; you should first understand the solution implementation on your own (i.e., you are encouraged to ask conceptual questions regarding semantics; not questions related to implementation of those concepts).

We have the same assumptions as in HW4:

1. Any variable used in an expression will be either declared before that expression or provided with some valuation in the initial environment.
2. Declaration of variables with same name in the same block context will not be present in any valid program.

The semantic rules for block contexts continue to hold in our new language. The newly introduced semantics of function declarations and applications (calling a function) follows the standard semantics.

The input and output environment will be represented by the grammar:

```
Env   ->  () | (Els)
Els   ->  El | El Els
El    -> (Var FName) | (Var Number) | ((FName ParamList) (SSeq))
```

The semantic rules for functions (named functions and anonymous functions) are presented below:

1. Function declaration captures the name of the function, the parameter list and its definition as sequence of statements. The functions declared in a block are accessible in that block and any block nested in it.
2. Signature of a function is its name and the number of formal parameters of that function. The program may have multiple function declarations with same signature; the restriction being such declarations are not present in the same block context.
3. If there are two function declarations with same signature in blocks b_1 and b_2, where b_2 is nested in b_1, then any call to the function inside b_2 *after the function declaration* will result in the execution of the function declared inside b_2.
4. Function call captures the name of the function and the argument list. The semantics of a call is the semantics of the definition of the function where each parameter to the function is mapped to the valuation of the corresponding argument.

5. You can assume that for any function call, the corresponding function will be either declared before the call or will be present in the initial environment.

6. Arguments of the function call are evaluated in the block context where the function call is performed.

7. Functions have three types of variables: parameters, which are mapped to the valuation of the actual arguments based on the call to the function; local variables, which are declared inside the function definition; and other variable, which are neither parameters nor local. For the last type of varibles, we will follow *dynamic scoping*, if the last argument for the call-statement is 1; otherwise, we will follow *static scoping*. You can assume that no matter what scoping rule is used, variables are defined before being used. Remember, parameters and local variables are not accessible outside the function block.

8. The function names and variables are both considered as symbols in our language. As a result, it is possible to assign function names to variables (see the grammar for `Env`). Such assignment will not involve any arithmetic operation and variables with valuation equal to some function name will not be used in any arithmetic expression as well. A variable can be used as a function name at the time of function invocation, if it is mapped to some function name. You can assume that in such scenario, the variable being used as function name will be assigned to a valid function name prior to the call statement.

9. We have added a new type of arith-expression that represent anonymous functions. An anonymous function has no name, it may have some parameters, an arith-expression describing its body, and is associated with some arguments (the number of arguments is equal to the number of parameters). The semantics of an anonymous function is a number resulting from the evaluation of the arith-expression (representing the body of the function) in the context of an environment which includes the mapping between the formal parameters and the associated valuation of actual arguments. For instance,

   ```
   (
     (decl x)
     (assign x (anonf ((y) (+ y 1)) (x)))
   )
   ```

   results in incrementing the value of `x` by 1 using the following semantics:

   `(anonf ((y) (+ y 1)) (x))` has the formal parameter `y`, actual argument `x`. The valuation of `x` is 0, which implies the valuation of `y` is 0. Therefore, the semantics of the body of the anonymous function is 1.

   Anonymous functions are interpreted following dynamic scoping rules.

Programming Rules

- You are required to submit one file hw5-⟨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, operations on numbers—in short, only the constructs we have covered in class. No imperative-style constructs, such as begin-end or explicit 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.

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

  ```
  (define p0
    '(
       (fundecl (f (x)) (
                           (assign y (+ x 1)))
                        )
       (decl y)
       (call (f (0)) 1)
     )
  )
  ;; expected result for '() input environment is
  ;; '((y 1) ((f (x)) ((assign y (+ x 1)))))


  (define p1
    '(
       (fundecl (f (x)) (
                           (assign y (+ x 1)))
                        )
  ```

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

```
      (decl y)
      (decl z)
      (assign z f)
      (call (z (0)) 1)
  )
)
;; expected result for '() input environment is
;; ((z f) (y 1) ((f (x)) ((assign y (+ x 1)))))

(define p2
  '(
      (decl y)
      (decl z)
      (assign z f)
      (call (z (0)) 1)
   )
)
;; expected result for '(((f (x)) ((assign y (+ x 1)))))
;; ((z f) (y 1) ((f (x)) ((assign y (+ x 1)))))


(define p3
  '(
     (decl x)
     (fundecl (f (z)) (
                       (assign x (+ z 1))
                       )
              )
     (fundecl (g (y)) (
                       (decl x)
                       (call (f (y)) 0)
                      )
              )
     (call (g (2)) 1)
     ))

;; expected result
;; > (sem p3 '())
;; '(((g (y)) ((decl x) (call (f (y)) 0)))
;;   ((f (z)) ((assign x (+ z 1))))
;;   (x 3))
```

```
(define p4
  '(
    (decl x)
    (fundecl (f (z)) (
                      (assign x (+ z 1))
                      )
             )
    (fundecl (g (y)) (
                      (decl x)
                      (call (f (y)) 1)
                     )
             )
    (call (g (2)) 1)
    ))

;; expected result
;;> (sem p4 '())
;; '(((g (y)) ((decl x) (call (f (y)) 1)))
;;   ((f (z)) ((assign x (+ z 1))))
;;   (x 0))

(define p5
  '(
    (decl x)
    (fundecl (f (z)) (
                      (assign x (+ z 1))
                      )
             )
    (fundecl (g (y)) (
                      (call (f (y)) 1)
                      )
             )
    (fundecl (h (w)) (
                      (decl x)
                      (call (g (w)) 0)
                      )
             )
    (call (h (10)) 0)
    ))
;; expected result
;;> (sem p5 '())
;;   '(((h (w)) ((decl x) (call (g (w)) 0)))
;;      ((g (y)) ((call (f (y)) 1)))
;;      ((f (z)) ((assign x (+ z 1))))
;;      (x 11))
```

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.