Homework: Optional Assignment 2

Due-date: Apr 14 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.

This is an optional assignment. If you do this assignment, the points you get for this assignment will be considered as extra-credits, and will add at most 8 *points to your overall grade.*

Learning Outcomes

- Knowledge and application of Functional Programming
- Ability to understand grammar specification
- Ability to design software following requirement specifications (verification by Model Checking)

Questions

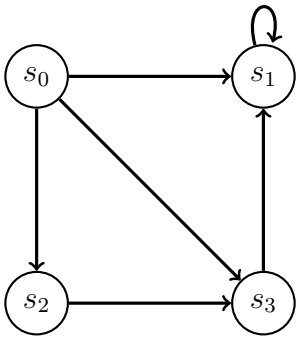
1. We define graph $G = (V, E, L)$, where V is the set of vertices in the graph, E is the set of directed edges between vertices and L is a mapping function, which associates each vertex with a list of propositions. For this problem, we further assume that each vertex has at least one out-going edge. Such a graph is represented in Racket as a list of vertices along with their L-mapping, and as a list of directed edges.

   For instance,

```
(define sts
  '( (s0 (p q))
     (s1 (p))
     (s2 (q))
     (s3 ())))

(define trs
  '( (s0 s1)
     (s0 s2)
     (s0 s3)
     (s1 s1)
     (s2 s3)
     (s3 s1)
    ))
```

corresponds to the following graph, where the vertex s_0 is associated with set of propositions $\{p, q\}$, vertex s_1 is associated with set of propositions $\{q\}$, vertex s_2 is associated with set of propositions $\{q\}$, and s_3 is associated with an empty set of propositions.



We define an expression language such that the semantics of that language is the set of vertices of a given graph (in the above form). The grammar (in Racket format) for the expression language is as follows:

```
Expr ->  tr | fa | (prop Var)
       | (not Expr) | (and Expr Expr) | (or Expr Expr)
       | (ax Expr) | (ex Expr)
       | (starax Expr) (starex Expr)
Var  ->  symbol
```

In the above, `Var` is any Racket symbol. Examples of valid expressions are

```
(define e1 'tr)
(define e2 'fa)
(define e3 '(or (prop x) (prop y)))
(define e4 '(starax (starex (or (prop x) (not (ex (prop y)))))))
```

The semantic function for the language generated by the above grammar takes as input

- a valid expression (as per above grammar in Racket; assume your input will never be invalid)
- a list of vertices (with mappings in Racket list format)
- a list of transitions (in Racket list format)

and will output a list of vertices following the semantic rules.

The semantic rules are

(a) The semantics of `tr` is the list of all vertices.

(b) The semantics of `fa` is an empty list.

(c) The semantics of `(prop x)` is the list of vertices, each of which are associated to `x`. For instance, the semantics of `(prop p)` for the above graph is `(s0 s1)`.

(d) The semantics of `(not Expr)` is the list of vertices that are not in the result of the semantics of `Expr`.

(e) The semantics of `(and Expr1 Expr2)` is the list of vertices that are both in the result of the semantics of `Expr1` and `Expr2` (i.e., intersection of two sets).

(f) The semantics of `(or Expr1 Expr2)` is the list of vertices that are in at least one of the results of the semantics of `Expr1` and `Expr2` (i.e., union of two sets).

(g) The semantics of `(ex Expr)` is the list of vertices, each of which has at least one directed edge to some vertex in the result of the semantics of `Expr`.

(h) The semantics of `(ax Expr)` is identical to the semantics of `(not (ex (not Expr)))`.

(i) The semantics of `(starex Expr)` is computed iteratively as follows:

Step 1: compute the semantics of `Expr`. Let us refer to the semantics as V_0 (list of vertices). Let $i = 0$.

Step 2: Repeat the following steps

a: Find the set of vertices V, each of which has at least one directed edge to a vertices in V_i.

b: Union V and V_i to get V_{i+1} and increment i to $i + 1$

Until V_i and V_{i-1} are identical

Step 3: return V_i

(j) The semantics of `(starax Expr)` is computed iteratively as follows:

Step 1: compute the semantics of `Expr`. Let us refer to the semantics as V_0 (list of vertices). Let $i = 0$.

Step 2: Repeat the following steps

a: Find the set of vertices V such each all directed edges from any vertex in V leads to some vertex in V_i.

b: Union V and V_i to get V_{i+1} and increment i to $i + 1$

Until V_i and V_{i-1} are identical

Step 3: return V_i

You are required to write the semantic function `vsem` following the above rules. For instance,

```
> (vsem 'tr sts trs)
'(s0 s1 s2 s3)
;; as the semantics of tr is the list of all vertices
```

```
> (vsem 'fa sts trs)
'()
;; as the semantics of fa is an empty list

> (vsem '(or (prop p) (prop q)) sts trs)
'(s0 s1 s2)
;; semantics of (prop p) is (s0 s1)
;; semantics of (prop q) is (s0 s2)

> (vsem '(ex (prop p)) sts trs)
'(s0 s1 s3)
;; semantics of (prop p) is (s0 s1)
;; s0 has an edge to s1
;; s1 has an edge to s1
;; s3 has an edge to s1
;; Therefore, the result is (s0 s1 s3)

> (vsem '(ax (prop p)) sts trs)
'(s1 s3)
;; semantically equivalent to (not (ex (not (prop p))))
;; semantics of (prop p) is (s0 s1)
;; semantics of (not (prop p)) is (s2 s3)
;; semantics of (ex (not (prop p))) is (s0 s2)
;; semantics of (not (ex (not (prop p)))) is (s1 s3)

> (vsem '(starex (prop p)) sts trs)
'(s2 s3 s0 s1)
;; semantics of (prop p) is (s0 s1) = V0
;; Iterations:
;;  list of vertices that has at least one edge to V0
;;     is (s0 s1 s3). This union V0 = V1 = (s0 s1 s3)
;;  V1 is not equal to V0
;;  list of vertices that has at least one edge to V1 union V0
;;     is (s0 s1 s2 s3). This union V1 = V2 = (s0 s1 s2 s3)
;;  V2 is not equal to V1
;;  list of vertices that has at least one edge to V2
;;     is (s0 s1 s2 s3). This union V2 = V3 = (s0 s1 s2 s3)
;;  V3 is equal to V2
;; return V3
```

```
> (vsem '(starax (prop p)) sts trs)
'(s0 s1 s2 s3)
;; semantics of (prop p) is (s0 s1) = V0
;; Iterations:
;;  list of vertices such that all their outgoing edges lead to
;;  some vertices in V0:  (s1 s3). This union V0 = V1 = (s0 s1 s3)
;;  V1 is not equal to V0
;;  list of vertices such that all their outgoing edges lead to
;;  some vertices in V1:  (s0 s1 s3). This union V1 = V2 = (s0 s1 s3 s2
;;  V2 is not equal to V1
;;  list of vertices such that all their outgoing edges lead to
;;  some vertices in V2: (s0 s1 s2 s3).
;;   This union V2 = V3 = (s0 s1 s2 s3)
;;  V2 is equal to V3
;; return V3
```

The ordering of the vertices in the output list is of no consequence.

Programming Rules

- You are required to submit one file hwopt2-⟨net-id⟩.rkt[1]. The file must start with the following.

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

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

- 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.

- The focus is on correct implementation. In this assignment, we will not assess the efficiency; however, avoid re-computations and use tail-recursion, if possible.

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. 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.
6. Starting two days before the deadline to do the above is likely to make it impossible for you to complete this assignment.
7. 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.

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

Postscript. It is not necessary to read this for successfully completing the assignment. This section is intended to provide with some additional context/knowledge. So far you have worked on functionally solving problems that relate to language semantics, job scheduling and simple regression.

In this optional assignment, you are computing the semantics of a type of logic: computation tree temporal logic. Such a logic is typically used to verify properties of discrete time dynamic systems (systems that evolve over time, where the actual/real-time necessary for the evolution is abstracted), e.g., certain types of software and hardware systems. The dynamics of the systems are described using the graph presented in this assignment, where the vertices represent the configurations of the system and the directed edges describe the evolution of the system from one configuration to another. The computation of semantics is referred to as model checking and forms the starting point for R&D in formally verifying and validating dynamic systems. For instance, one can prove/disprove (have disproved/proved) the correctness of Java's synchronization protocol, public key encryption protocols, safety properties of Curiosity Rover[2]. If you are interested, you can consider taking courses such as `AER E/COM S 407X` (taught by Aerospace Engineering) and/or `COM S/SE/CPR E 412` (taught by Computer Science).

[2]https://www.usenix.org/conference/hotdep12/workshop-program/presentation/holzmann