Exam 1: COMS/SE 342

February 23, 2017

Time: 70 mins

____

Last Name First Name

Learning Outcomes

- Application of knowledge of computing and mathematics
- Ability to understand the implications of mathematical formalisms in computer science
- Ability to understand CFG specifications and semantic evaluation
- Ability to encode functional definitions

You can consult lecture materials, review publicly available online resources, and Racket to answer any questions in this test. You are not allowed to collaborate in any form with anyone[1].

Question	Points	Score
1 (Context-free Grammar)	12	
2 (Lambda Calculus)	10	
3 (Racket: Complete Racket Code)	4	
4 (Racket: Write Racket Code)	6	
5 (Operational Semantics)	8	
Total	40	

[1]Points will not be deducted for minor syntactic errors in Racket Questions. Writing car instead of cdr is not a syntactic error.

Questions

1. Answer True/False and justify your answer.

   (a) A grammar always generates language containing infinite number of strings.

   (b) All palindromes (a string that reads the same forward and backward, including empty string ϵ) over the letters a and b are generated by the following grammar (S is the start symbol, and a, b and ϵ are terminals)

$$S \rightarrow aSa \mid bSb \mid \epsilon$$

(c) In the following grammar,

$$\begin{aligned} bar &\longrightarrow baz \\ foo &\longrightarrow bar \ \$ \ foo \\ baz &\longrightarrow (\ foo\ ) \\ bar &\longrightarrow bar \ \# \ baz \\ baz &\longrightarrow x \\ baz &\longrightarrow y \\ foo &\longrightarrow bar \end{aligned}$$

where the terminals are $\{\$, \#, (,), x, y\}$, foo is the start non-terminal, and $\$, \#$ are some binary operators

i. $\$$ has higher precedence than $\#$.

ii. The grammar cannot generate the string $x\$x\#y\#(y\$x)$.

2. Consider the following λ-expressions:

$$Y: \ \lambda t.(\lambda x.(t\ (x\ x))\ \ \lambda x.(t\ (x\ x)))$$

$$k: \ \lambda f.\lambda n.(\text{if } (equal?\ n\ 0)\ \ 1\ \ (\times\ n\ (f\ (-\ n\ 1))))$$

The semantics of if-statement above is identical to the one you used in Racket. For instance, in (if $(equal?\ n\ 0)\ \ 1\ \ (\times\ n\ (f\ (-\ n\ 1))))$, when the condition $(equal?\ n\ 0)$ is true, the value 1 is returned, otherwise evaluate $(\times\ n\ (f\ (-\ n\ 1)))$.

Compute: $((Y\ k)\ 2)$.

3. Consider the following Racket function: minmax, which has one formal parameter. The parameter is a list of numbers. The function computes the smallest and the largest numbers in the list, and outputs a list containing these numbers. The following examples present instances of application of the minmax function

   ```
   > (minmax '())
   '()

   > (minmax '(1))
   '(1)

   > (minmax '(1 2 3 4))
   '(1 4)

   > (minmax '(2 1 4 3))
   '(1 4)
   ```

   The following is an incomplete definition of the function minmax.

   ```
   (define (minmax lst)
     (if (< (length lst) <A>)
          <B>
          (if (equal? (length lst) 2)
               (if (< (car lst) (cadr lst))
                    <C>
                    <D>
               (if (< (car lst) (car (minmax (cdr lst))))
                    <E>
                    (if (> (car lst) (cadr (minmax (cdr lst))))
                         <F>
                         <G> )))))
   ```

   Complete the definition by replacing the following:

   (a) `<A>`
   (b) `<B>`
   (c) `<C>`
   (d) `<D>`
   (e) `<E>`
   (f) `<F>`
   (g) `<G>`

4. Consider a function `myfunc` with two arguments A_1 and A_2 such that

   - A_1 is a conditional function (predicate) which takes one argument of type T and returns either true or false
   - A_2 is a list of elements of type T

   The function returns a list containing only the elements in A_2 for which A_1 returns true.

   For instance, given

```
(define (isoperator x)
   (or (equal? x '+) (equal? x '-)))
```

   the following illustrates the usage of `isoperator` as A_1 and `1 + 2 + 3 - 5` as A_2 in `myfunc`

```
> (myfunc isoperator '(1 + 2 + 3 - 5))
'(+ + -)
```

   Similarly,

```
> (myfunc number? '(1 + 2 + 3 - 5))
'(1 2 3 5)

> (myfunc even? '(1 2 3 5))
'(2)
```

   Write the definition of function `myfunc`. (*You are only allowed to use the basic list operations: car, cdr, cons, list, null?, list?.*)

5. We are given a grammar for propositional logic expressions with the logical operators: not and $\#$, and with the following production rules:

$$Expr \;\rightarrow\; AP \mid (not\ Expr) \mid (\#\ Expr\ Expr)$$

For instance, the grammar can generate expressions such as p, $(not\,p)$, $(\#\,p\,q)$, $(\#\,(not\,p)\,q)$; where p and q are atomic propositions (AP).

We are also given the semantics of the expressions (generated by the above grammar) in the context of an environment (say, σ), which captures whether a proposition value is true or false. The value of any proposition p as per a σ is denoted by $\sigma(p)$. For instance, if $\sigma = ((p\ true)\ (q\ false))$ is the environment, then $\sigma(p) = true$, $\sigma(q) = false$ and $\sigma(r)$ is unknown. The semantic rules are as follows:

$$[\![AP]\!]_\sigma = \sigma(AP)$$

$$[\![(not\ Expr)]\!]_\sigma = \begin{cases} false & \text{if } [\![Expr]\!]_\sigma = true \\ true & \text{otherwise} \end{cases}$$

$$[\![(\#\ Expr_1\ Expr_2)]\!]_\sigma = \begin{cases} true & \text{if } \textbf{only one of } [\![Expr_1]\!]_\sigma \text{ and } [\![Expr_2]\!]_\sigma \text{ is equal to } false \\ false & \text{otherwise} \end{cases}$$

(a) Compute $[\![(\#\ (not\ \ p)\ \ (not\ \ q))]\!]_\sigma$ where $\sigma = ((p\,true))$

(b) Compute $[\![(\#\ (not\ \ p)\ \ (\#\ (not\ \ p)\ \ q))]\!]_\sigma$ where $\sigma = ((q\ true))$

Exam 2: COMS/SE 342

April 4, 2017

Time: 70 mins

____

Last Name First Name

Learning Outcomes

- Application of knowledge of computing and mathematics
- Ability to understand the implications of mathematical formalisms in computer science
- Ability to understand CFG specifications and semantic evaluation
- Ability to encode functional definitions
- Ability to understand the concepts of function invocations & references

You can consult lecture materials, review publicly available online resources, and Racket to answer any questions in this test. You are not allowed to collaborate in any form with anyone[1].

Question	Points	Score
1 (Racket: Code Comprehension)	3	
2 (Racket: Code Augmentation)	7	
3 (Lambda Calculus)	10	
4 (Functions/Recursion)	6	
5 (References with Functions)	14	
6 (Extra Credit: Arrays)	10	
Total	40	

[1]Points will not be deducted for minor syntactic errors in Racket Questions. Writing car instead of cdr is not a syntactic error.

Questions

1. Stuart and Jerry are asked to write a function that takes as input a list and a number n and divides the list into two partitions, the first partition contains first $\lfloor \frac{n}{2} \rfloor$ elements in the list and the second partition contains the rest of the elements in the list. The following shows the version written by Stuart and Jerry.

```
;; Stuart's version;;;;;
(define (divide lst n)
  (if (< n 2)
      (list '() lst)
      (list (cons (car lst) (car (divide (cdr lst) (- n 2))))
            (cadr (divide (cdr lst) (- n 2))))))
;;;;;;;;;;;;;;;;;;;;;;;;

;; Jerry's version;;;;;
(define (divide lst n)
  (if (< n 2)
      (list '() lst)
      (createpair (car lst) (divide (cdr lst) (- n 2)))))

(define (createpair x pair)
  (list (cons x (car pair)) (cadr pair)))
;;;;;;;;;;;;;;;;;;;;;;;;
```

Scarlet Overkill reviewed the two versions and claimed that divide function written by Jerry incurs less computation overhead in terms of number of function calls compared to the ones written by Stuart. Do you agree or disagree with Miss Overkill? Justify your answer.

2. Assume that you are given a function `merge` that takes as input two sorted (ascending order) list of numbers and outputs a list where all the numbers from the two lists are arranged in ascending order. For instance, the following shows the usage of the `merge` function

   ```
   > (merge '(1 3 5) '(2 4 6))
   '(1 2 3 4 5 6)
   > (merge '(1 3 5) '())
   '(1 3 5)
   > (merge '(1 3 5) '(1 3 5))
   '(1 1 3 3 5 5)
   >
   ```

   Write a function `sort` that takes as input list of numbers and returns a list, where all the numbers from the input list are sorted in ascending order. You **must use** the functions `merge` (*You are not required to write the function* `merge`.), the function `divide` (from previous question), and you are allowed to use basic list operations and conditionals such as `car, cadr, cdr, cons, list, length, list?, null?`, boolean operators such as `or`, `and` and `not`, and control constructs such as recursion and if-then-else. You must not use or write any additional helper functions.

3. Consider the following lambda expressions:

$$\begin{aligned} &P : \lambda a.\lambda b.\lambda f.((f\ a)\ b) \\ &Q : \lambda g.(g\ \lambda a.\lambda b.a) \\ &R : \lambda g.(g\ \lambda a.\lambda b.b) \\ &S : \lambda x.\lambda a.\lambda b.a \end{aligned}$$

Compute the result of the following:

(a) Compute $(Q\ ((P\ 1)\ ((P\ 2)((P\ 3)\ S))))$

(b) Compute $(R\ ((P\ 1)\ ((P\ 2)((P\ 3)\ S))))$

(c) Compute $(Q\ (R\ (R\ ((P\ 1)\ ((P\ 2)((P\ 3)\ S))))))$

4. Recall the grammar you have used in homework assignment 6, where we used two meta-syntax rules for validating the programs. The following production rules of the grammar are the same as the one used in the assignment.

```
Program      -> Expr
Expr         -> Number | Variable | OpExpr | FExpr | ApplyF
OpExpr       -> ArithExpr | CondExpr | VarExpr
ArithExpr    -> (Op Expr Expr)
Op           -> + | -  | * | /
CondExpr     -> (CCond Expr Expr)
CCond        -> BCond | (or CCond CCond) | (and CCond CCond) | (not CCond)
BCond        -> (gt Expr Expr) | (lt Expr Expr) | (eq Expr Expr)
VarExpr      -> (var VarAssign Expr)
VarAssign    -> (VarAssignSeq)
VarAssignSeq -> (Variable Expr) | (Variable Expr) VarAssignSeq
Variable     -> symbol
FExpr           -> (fun FAssign Expr)
FAssign         -> ((FName FormalParams) Expr)
FormalParams    -> () | (FormalParamList)
FormalParamList -> Variable | Variable FormalParamList
ApplyF          -> (apply (FName Args))
Args            -> () | (ArgsList)
ArgList         -> Expr | Expr ArgList
```

A program written in the above language is syntactically correct if and only if it follows the grammar rules as well as the meta-syntax rules.

A programmer wants to use the language following the above grammar to write two types of recursions necessary for developing some software system.

- *Type I recursion*: A function calls itself in its definition
- *Type II recursion*: A function F calls another function G in its definition and G calls function F in its definition.

The language designer who wrote the above grammar assures the programmer that both Type I and Type II recursions can be used in syntactically correct programs written in the language. Vector Perkins claims that the language designer is wrong. Do you agree or disagree with Vector? Justify your answer (if you agree explain why at least one of the Type I and Type II recursions cannot be correctly implemented in the above language; if you disagree present examples of both Type I and Type II recursions correctly represented in the above language).

5. Recall that the language we developed as part of the course is close to C/C++. Consider the following program:

```
void f(int* i) {
   *i = 21;          // update the pointee to 21
                     // i.e., write at location pointed to by i
}

int main() {
   int* x = new int; // allocating space
   *x = 11;          // pointee of pointer x is 11
   f(x);           // call function
   cout << *x;       // print pointee of pointer x
   return 0;
}
```

This is equivalent to the program written in our language:

```
(fun    ( (main ())
            ;; definition of function f
            (fun ( (f (i))
                   (wref i 21))  ;; write to the location i
            ;;
            ;; sequence of operations in main
              (var (x (ref 11))  ;; allocate memory and write to it 11
                                 ;; x holds the value of the location
                  (var (y (apply (f (x)))) ;; call the function
                       (deref x))) ;; output the value at the location x
        )
 (apply (main ()))
)
```

(a) Assuming there is at least one *free* location in our heap, what is the semantics of the program. Justify your answer.

(b) Now consider the following program.

```
void f(int* i) {
   *i = 21;          // update the pointee to 21
                     // i.e., write at location pointed to by i
}

int main() {
   int* x = new int; // allocating space
   *x = 11;          // pointee of pointer x is 11
   int* y = x;
   f(y);             // call function
   cout << *x;       // print pointee of pointer x
   return 0;
}
```

i. Translate the above program to an equivalent program in our language.

ii. What is the semantics of the translated program, assuming there is at least one free location in our heap? Justify your answer.

6. **[EXTRA CREDIT 10pts]** We are going to add few new expressions to the language we have been developing as part of this course. They are declaration of arrays of specific size, and reading from and writing to specific indices in the array.

```
Expr  -> ... | (arr (NameofArray Size) Expr)
             | (read (NameofArray Expr))
             | (write (NameofArray Expr Expr))
NameofArray -> symbol
Size -> Number
```

There are few meta-syntax rules we will follow. All array names used in the program are distinct. The expressions in the read and write operations do not involve heap-operations.

Recall that the semantics is represented as a value/exception paired with the heap. That is, $[\![Expr]\!]^h_\sigma = (v\ h')$. We will use the notation: $val([\![Expr]\!]^h_\sigma) = v$ and $heap([\![Expr]\!]^h_\sigma) = h'$.

The semantics of the new expressions are given as follows:

$$[\![\texttt{(arr (Name Size) Expr)}]\!]^h_\sigma = \begin{cases} [\![\texttt{Expr}]\!]^h_{\texttt{(arr (Name Size) L)} \circ \sigma} \\ \quad \text{if } \exists \texttt{L}.\forall i.\texttt{L} \leq i \leq \texttt{Size} + \texttt{L} \Rightarrow (h(i) = free) \\ ((\texttt{exception oom})\ h) \quad \text{otherwise} \end{cases}$$

$$[\![\texttt{(read (Name Expr))}]\!]^h_\sigma = \begin{cases} (h(i)\ h) \quad \text{if } \texttt{(arr (Name Size) L)} \in \sigma \wedge \\ \qquad i = \texttt{L} + val([\![\texttt{Expr}]\!]^h_\sigma) \\ ((\texttt{exception arr-not-found})\ h) \quad \text{otherwise} \end{cases}$$

$$[\![\texttt{(write (Name Expr1 Expr2))}]\!]^h_\sigma = \begin{cases} (val([\![\texttt{Expr2}]\!]^h_\sigma)\ h') \quad \text{if } \texttt{(arr (Name Size) L)} \in \sigma \wedge \\ \qquad h' = h/h(\texttt{L} + val([\![\texttt{Expr1}]\!]^h_\sigma)) \mapsto val([\![\texttt{Expr2}]\!]^h_\sigma) \\ ((\texttt{exception arr-not-found})\ h) \quad \text{otherwise} \end{cases}$$

Two important properties ensure the integrity of an array structure.

(a) If v is written to an array index of an array named $myarr$, then subsequent read operation involving the same index of the same array without any intermediate writes to the same array will produce v (paired with some heap).

(b) If v is read from an array index of an array named $myarr$, then it must the result of some preceding write operation to the same index of the same array.

Are any of the above properties violated by the semantics? Justify your answer.

Exam 2-Extra Credit Opt: COMS/SE 342

April 13, 2017

Time: 60 mins

____

Last Name First Name

The score for this exam can increase your exam 2 score by at most 10 points

$\boxed{\text{A}}$

Learning Outcomes

- Application of knowledge of computing and mathematics
- Ability to understand the implications of mathematical formalisms in computer science
- Ability to encode functional definitions
- Ability to understand CFG specifications and semantic evaluation

You can consult lecture materials, review publicly available online resources, and Racket to answer any questions in this test. You are not allowed to collaborate in any form with anyone[1].

Question	Points	Score
1 (Racket)	10	
2 (Lambda Calculus)	10	
3 (Language Semantics)	10	
Total	30	

[1]Points will not be deducted for minor syntactic errors in Racket Questions. Writing car instead of cdr is not a syntactic error.

Questions

1. A string S is a subsequence of another string T, if the former can be obtained from the latter by deleting some characters from the T (while keeping the remaining in order). In our case, we will represent strings as lists and characters in strings as elements in the list. For instance, `(a t g c t a g)` is subsequence of `(t a c t g c t a t g t)`.

   (a) Write a function `subseq` that can be applied to two lists and it returns true if and only if the first list is a subsequence of the list.

   (b) Write a function `countif` that can be applied to three parameters where

   - first parameter is any function f which can be applied on two lists and returns either true or false.
   - second parameter is a list
   - third parameter is a list of lists

   The countif returns the number of times the f returns true, when applied to second parameter and each element of third parameter. For example, (countif subseq '(a t g) '((a g t) (a t g a) (a a t g))) returns 2.

2. Given

$$\begin{aligned}
&P : \lambda a.\lambda b.a \\
&Q : \lambda a.\lambda b.b \\
&R : \lambda c.((c\, Q)\ P)
\end{aligned}$$

Compute

(a) $(R\, (R\, (R\, P)))$

(b) $(((\lambda z1.\lambda z2.\lambda z3.((z1\ z2)\ z3)\ (R\, Q))\ P)\ r2)$

3. As part of the assignments, we have designed a simple language and its interpreter.

   (a) Consider the case, where you are required to add a new construct to this language of the form `(seq Expr1 Expr2)`. The semantics of seq-expression is the semantics of `Expr2`, when evaluated right after `Expr1`. For instance, `(seq (ref 10) (ref 20))` using a heap `((1 free) (2 free))` evaluates to `(2 ((1 10) (2 20)))`. Comment on whether the existing semantic-rules are **sufficient** to express the semantics of the seq-expression. Justify your answer.

   (b) In the language we have developed so far, we have considered just one data type: number. Consider the case, where you are required to add another data type: boolean. Comment on the changes in the semantics that may be **necessary** to incorporate this addition. Justify why you think these changes are necessary.