Definition of Programming Languages

CPSC 311 2016W1

University of British Columbia

Practice Midterm Examination
26 October 2016, 19:00–21:00

Joshua Dunfield

Student Name:		Student Number:	
Signature:		CS userid:	

INSTRUCTIONS

- This is a CLOSED BOOK / CLOSED NOTES examination.
- This examination is PRINTED ON BOTH SIDES of the paper.
- Write all answers ON THE EXAMINATION PAPER.

  Try to write your answers in the blanks or boxes given. If you can't, try to write them elsewhere on the same page, or on one of the worksheet pages, and **LABEL THE ANSWER**. We can't give credit for answers we can't find.
- Blanks are suggestions. You may not need to fill in every blank.
- This examination is expected to be worth 15% of the course grade.

Notes for the practice exam:

- The real questions will be **on the same general topic** as the midterm questions. We are trying to maintain similar difficulty levels, erring on the side of making the practice questions slightly harder than the actual midterm questions.
- The practice midterm has silly footnotes. The actual midterm doesn't (it may have silly question titles, though).
- Point totals may not correspond exactly to the actual midterm.

Worksheet α

Question 1 [15 points]: A substitutable fox

A fox has wandered into the Mathematics Annex, and has brought its own definition of substitution. Fortunately, most of the definition is the same as what you've seen before. It appears that foxes are abstract syntax.

$$\begin{aligned}
[e2/x](\mathsf{Id}\ x) &= e2 \\
[e2/x](\mathsf{Id}\ y) &= (\mathsf{Id}\ y) \qquad \text{if } x \neq y \\
[e2/x](\mathsf{Fox}\ eL\ eR) &= \big(\mathsf{Fox}\ \ [e2/x]eL\ \ [e2/x]eR\big) \\
[e2/x](\mathsf{Let}\ x\ e\ eB) &= \big(\mathsf{Let}\ \ x\ \ [e2/x]e\ eB\big) \\
[e2/x](\mathsf{Let}\ y\ e\ eB) &= \big(\mathsf{Let}\ \ y\ \ [e2/x]e\ \ [e2/x]eB\big) \\
&\quad \text{if } x \neq y
\end{aligned}$$

Following the above definition, fill in the blank:

$$\Big[(\mathsf{Id}\ m)\Big/z\Big]\big(\mathsf{Fox}\ (\mathsf{Id}\ z)\ (\mathsf{Let}\ z\ (\mathsf{Id}\ z)\ (\mathsf{Id}\ z))\big)$$

$$= \Big(\mathsf{Fox}\ \ [(\mathsf{Id}\ m)/z](\mathsf{Id}\ z)\ \ [(\mathsf{Id}\ m)/z]\big(\mathsf{Let}\ z\ (\mathsf{Id}\ z)\ (\mathsf{Id}\ z)\big)\Big)$$

$$= \Big(\mathsf{Fox}\ \ (\mathsf{Id}\ m)\ \ \big(\mathsf{Let}\ z\ [(\mathsf{Id}\ m)/z](\mathsf{Id}\ z)\ (\mathsf{Id}\ z)\big)\Big)$$

$$= \Big(\mathsf{Fox}\ \ (\mathsf{Id}\ m)\ \ \big(\mathsf{Let}\ z\ (\mathsf{Id}\ m)\ (\mathsf{Id}\ z)\big)\Big)$$

It's not mandatory to write the intermediate steps. But you're more likely to get the right answer if you do—and you're more likely to get partial credit for a wrong answer.

You might be wondering how to evaluate a fox. That's a good question. But you don't need to know how to evaluate a fox to fill in the blank above, because substitution doesn't do evaluation.

Question 2 [30 points]: Energy

This question is about a peculiar recursion expression called B-Rec. The b stands for *bounded* recursion, or perhaps for *battery-powered* recursion.

In place of the Fun recursion expression $(\mathsf{Rec}\ u\ e)$, we write $(\mathsf{B\text{-}Rec}\ k\ u\ e)$ for an expression that can nest itself *up to* k *times*, where k is a natural number $(0, 1, 2, \ldots)$.

This recursion expression will not cause the interpreter to loop forever; instead, it will "give up" if its recursion depth exceeds k. For example, in $(\mathsf{B\text{-}Rec}\ 1\ u\ e)$, the body e can use u recursively—but those recursive uses of u must not be recursive themselves.

We enforce this in the evaluation rules. If $(\mathsf{B\text{-}Rec}\ k\ u\ e)$ tries to call itself more than k times, then it will not evaluate to a value. To represent that situation, we add a judgment e drained.

$$\frac{k \geq 1 \qquad k' = k - 1 \qquad \big[(\mathsf{B\text{-}Rec}\ k'\ u\ e)/u\big]e \Downarrow v}{(\mathsf{B\text{-}Rec}\ k\ u\ e) \Downarrow v}\ \text{Eval-b-rec} \qquad \frac{k < 1}{(\mathsf{B\text{-}Rec}\ k\ u\ e)\ \mathsf{drained}}$$

The premise $k \geq 1$ of Eval-b-rec prevents Eval-b-rec from deriving $(\mathsf{B\text{-}Rec}\ 0\ u\ e) \Downarrow v$.

You may need to use the following rules:

$$\frac{}{(\mathsf{Num}\ n) \Downarrow (\mathsf{Num}\ n)}\ \text{Eval-num} \qquad \frac{}{(\mathsf{Btrue}) \Downarrow (\mathsf{Btrue})}\ \text{Eval-btrue} \qquad \frac{}{(\mathsf{Bfalse}) \Downarrow (\mathsf{Bfalse})}\ \text{Eval-bfalse}$$

$$\frac{e \Downarrow (\mathsf{Btrue}) \qquad eThen \Downarrow v}{(\mathsf{Ite}\ e\ eThen\ eElse) \Downarrow v}\ \text{Eval-ite-true} \qquad \frac{e \Downarrow (\mathsf{Bfalse}) \qquad eElse \Downarrow v}{(\mathsf{Ite}\ e\ eThen\ eElse) \Downarrow v}\ \text{Eval-ite-false}$$

Q2a [15 points] Derive the judgment written below the line.

If you need it, use:

$$\Big[\Big(\mathsf{B\text{-}Rec}\ 0\ u\ \big(\mathsf{Ite}\ (\mathsf{Btrue})\ (\mathsf{Num}\ 3)\ (\mathsf{Id}\ u)\big)\Big)\Big/u\Big]\big(\mathsf{Ite}\ (\mathsf{Btrue})\ (\mathsf{Num}\ 3)\ (\mathsf{Id}\ u)\big)$$
$$= \Big(\mathsf{Ite}\ (\mathsf{Btrue})\ (\mathsf{Num}\ 3)\ \big(\mathsf{B\text{-}Rec}\ 0\ u\ \underbrace{(\mathsf{Ite}\ (\mathsf{Btrue})\ (\mathsf{Num}\ 3)\ (\mathsf{Id}\ u))}_{eBody}\big)\Big)$$

For convenience, you can write "eBody" instead of $\big(\mathsf{Ite}\ (\mathsf{Btrue})\ (\mathsf{Num}\ 3)\ (\mathsf{Id}\ u)\big)$.

$$\dfrac{1 \geq 1 \qquad 0 = 1 - 1 \qquad \dfrac{\dfrac{}{(\mathsf{Btrue}) \Downarrow (\mathsf{Btrue})}\ \text{Eval-Btrue} \qquad \dfrac{}{(\mathsf{Num}\ 3) \Downarrow (\mathsf{Num}\ 3)}\ \text{Eval-num}}{\Big(\mathsf{Ite}\ (\mathsf{Btrue})\ (\mathsf{Num}\ 3)\ \big(\mathsf{B\text{-}Rec}\ 0\ u\ eBody\big)\Big) \Downarrow (\mathsf{Num}\ 3)}\ \text{Eval-ite-true}}{\Big(\mathsf{B\text{-}Rec}\ 1\ u\ \big(\mathsf{Ite}\ (\mathsf{Btrue})\ (\mathsf{Num}\ 3)\ (\mathsf{Id}\ u)\big)\Big) \Downarrow (\mathsf{Num}\ 3)}\ \text{Eval-b-rec}$$

Question 2 [30 points]: Energy, continued

Here are the rules for B-Rec, again:

$$\frac{\mathsf{k} \geq 1 \qquad \mathsf{k}' = \mathsf{k} - 1 \qquad [(\text{B-Rec}\ \mathsf{k}'\ u\ e)/u]e \Downarrow v}{(\text{B-Rec}\ \mathsf{k}\ u\ e) \Downarrow v}\ \text{Eval-b-rec} \qquad \frac{\mathsf{k} < 1}{(\text{B-Rec}\ \mathsf{k}\ u\ e)\ \mathsf{drained}}$$

Q2b [15 points] Implement the above two rules by filling in the **type-case** branch for B-Rec in the function `interp`:

```
(define (interp e)   ; interp : E −> E
  (type-case E e
   ; ...
   [Ite (eCond eThen eElse)
     (let ([vCond (interp eCond)])
       (type-case E vCond
         [Btrue ()  (interp eThen)]
         [Bfalse () (interp eElse)]
         [else (error "Ite on a non-boolean")]))]

    [Rec (u eB)
     (let ([v (interp (subst (Rec u eB) u eB))])
       v)]

   [B-Rec (k u eB)
     (if (>= k 1)
         (let ([v (interp (subst (B-Rec (- k 1) u eB) u eB))])
           v)
         (error "B-Rec drained")
   )]
```

Under exam conditions, it's okay to follow the rules less closely; for example, by not let-binding v and just writing

```
(interp (subst (B-Rec (- k 1) u eB) u eB))
```

Question 3 [20 points]: “You could call it ‘Curry in a Hurry’.”[1]

Most real languages support functions (or procedures, or methods) that take more than one argument (or maybe zero arguments). Here, we’ll add functions that take **at least one** argument.

To add multi-argument functions, we can either add them “for real” by updating the abstract syntax and evaluation rules (and the typing rules, if our language is typed), or add them as syntactic sugar. In that case, we only need to change the parser.

In this problem, we’ll add multi-argument functions as syntactic sugar.

We need to update the concrete syntax of Fun, as follows:

```
⟨E⟩ ::= ...
      | {Lam ⟨Id⟩... ⟨E⟩}
      | {App ⟨E⟩ ⟨E⟩...}
```

The “...” are read as **one or more** occurrences of the preceding nonterminal. Thus, all of the following are syntactically valid expressions that match the above BNF:

```
{Lam x 5}          {App f 3}
{Lam x x}          {App g a}
{Lam x y 5}        {App {App 2 3} b 17}
{Lam x y z 5}      {App f 1 c 0}
```

However, `{Lam 5}` (and `{App f}`) are not syntactically valid: they would be functions (and applications) of zero arguments, which we won’t allow.

Since we’re implementing multi-argument functions as syntactic sugar, the abstract syntax of Fun doesn’t change. The relevant parts of the **define-type** are still

```
(define-type E
  [Num (n number?)]
  [Id (name symbol?)]
  [Lam (name symbol?) (body E?)]
  [App (function E?) (argument E?)]
)
```

Add code to `parse` to implement the `Lam` part of the following “desugaring”:

`{Lam x1 x2 ··· xn ⟨E⟩}` is parsed as `{Lam x1 {Lam x2 ··· {Lam xn ⟨E⟩} ···}}`

`{App ⟨E0⟩ ⟨E1⟩ ··· ⟨En⟩}` is parsed as `{App {···{App ⟨E0⟩ ⟨E1⟩} ···} ⟨En⟩}`

For example:

`{Lam x y {App z 5 y}}` is parsed as `{Lam x {Lam y {App {App z 5} y}}}`

so `(parse '{Lam x y {app z 5 y}})` should return the abstract syntax expression

$$\Big(\mathsf{Lam}\ x\ \Big(\mathsf{Lam}\ y\ \big(\mathsf{App}\ (\mathsf{App}\ (\mathsf{Id}\ z)\ (\mathsf{Num}\ 5))\ (\mathsf{Id}\ y)\big)\Big)\Big)$$

[1]Words uttered by a customer of India Garden in Pittsburgh, which the restaurant manager met with an icy glare.

Question 3 [20 points]: Curry in a Hurry, continued

```
(define (parse-lam args)
   (if (= (length args) 2)
       ; single-argument Lam
       (Lam (first args) (parse (second args)))

       ; multi-argument Lam
```

One approach (probably the easiest):

```
       (Lam (first args) (parse-lam (rest args)))
```

Alternate approach: create concrete syntax and pass that to parse.

```
       (let ([body  (parse (cons 'Lam (rest args)))])
         (Lam (first args) body))
  ))

(define (parse-app args)
  (if (= (length args) 2)
      ; App with one argument (note: (first args) is the function being applied)
      (App (parse (first args)) (parse (second args)))

      ; App with 2 or more arguments
      (let* ([last-arg    (last args)]
             [except-last (reverse (rest (reverse args)))])
        (App (parse-app except-last) (parse (last args))))
  ))

(define (parse sexp)
  (cond
    ; ...
    [(list? sexp)
     (let*
        ([head      (first sexp)]
         [args      (rest sexp)]
         [arg-count (length args)])

       (case head
        ; ...

         [(Lam) (if (< arg-count 2)
                    (error "parse: malformed `Lam'")
                    (if (symbol? (first args))
                        (parse-lam args)
                        (error "parse: Lam must be followed by an identifier"))
                  )]

         [(App) (if (< arg-count 2)
                    (error "parse: App needs at least two subexpressions")
                    (parse-app args))]
       ))]))
```

Worksheet β

Question 4 [15 points]: REDACTED

In this question, use the following evaluation rules:

$$\frac{}{(\mathsf{Num}\ n) \Downarrow (\mathsf{Num}\ n)}\ \text{Eval-num} \qquad \frac{}{(\mathsf{Lam}\ x\ e) \Downarrow (\mathsf{Lam}\ x\ e)}\ \text{Eval-lam}$$

$$\frac{e1 \Downarrow (\mathsf{Num}\ n1) \qquad e2 \Downarrow (\mathsf{Num}\ n2)}{(\mathsf{Add}\ e1\ e2) \Downarrow (\mathsf{Num}\ (n1 + n2))}\ \text{Eval-add} \qquad \frac{e1 \Downarrow (\mathsf{Num}\ n1) \qquad e2 \Downarrow (\mathsf{Num}\ n2)}{(\mathsf{Sub}\ e1\ e2) \Downarrow (\mathsf{Num}\ (n1 - n2))}\ \text{Eval-sub}$$

Recall the *expression strategy* and the *value strategy* for evaluating a function application $(\mathsf{App}\ e1\ e2)$. Both strategies evaluate $e1$ to $(\mathsf{Lam}\ x\ eB)$, but they differ in how they handle $e2$:

- The expression strategy evaluates eB with x replaced by $e2$.
- The value strategy evaluates $e2$ to a value $v2$, then evaluates eB with x replaced by $v2$.

Also recall the evaluation rules that define these strategies:

$$\frac{e1 \Downarrow (\mathtt{Lam}\ x\ eB) \qquad e2 \Downarrow v2 \qquad [v2/x]eB \Downarrow v}{(\mathtt{App}\ e1\ e2) \Downarrow v}\ \text{Eval-app-value}$$

$$\frac{e1 \Downarrow (\mathtt{Lam}\ x\ eB) \qquad [e2/x]eB \Downarrow v}{(\mathtt{App}\ e1\ e2) \Downarrow v}\ \text{Eval-app-expr}$$

The actual questions are redacted, because we couldn't figure out something that wouldn't give away the actual midterm questions.

However, we can say:

- the actual questions have the exact same preamble (the rules and text above), and
- they fit on the remainder of this page.

Question 5 [30 points]: Demonic Nondeterminism[2]

Some languages provide a feature that, given a list of expressions, evaluates one of them at random (well, pseudorandomly, if you want to get *precise* about it).

In this question, we extend the concrete syntax of the Fun language to include `Choose` blocks:

```
⟨E⟩ ::= ...
      | {Choose ⟨E⟩ ⟨E⟩...}
```

Here, ⟨E⟩... means zero or more occurrences, so a `Choose` contains one expression (the first ⟨E⟩) followed by zero or more expressions.

We also extend the `E` **define-type**, and add an evaluation rule:

```
(define-type E
  ⋮
  [Choose (expressions (listof E?))]
)
```

$$\frac{n \geq 1 \qquad 1 \leq k \leq n \qquad ek \Downarrow vk}{(\texttt{Choose}\ e1\ \cdots\ en) \Downarrow vk}\ \text{Eval-choose}$$

In Eval-choose, just one of the expressions, the kth expression, is evaluated. (Why just one? In the rule, k is a meta-variable, the same as n, so we can choose any integer k such that k is between 1 and n.)

However, this question deals with typing for `Choose` expressions, not with evaluation.

The typing rules reused from Fun that you may need for this question are:

$\boxed{\Gamma \vdash e : A}$ Under assumptions Γ, expression e has type A

$$\frac{\Gamma(x) = A}{\Gamma \vdash (\mathsf{Id}\ x) : A}\ \text{Type-var} \qquad \frac{}{\Gamma \vdash (\mathsf{Num}\ n) : \mathsf{num}}\ \text{Type-num}$$

$$\frac{}{\Gamma \vdash (\mathsf{Bfalse}) : \mathsf{bool}}\ \text{Type-false} \qquad \frac{}{\Gamma \vdash (\mathsf{Btrue}) : \mathsf{bool}}\ \text{Type-true}$$

$$\frac{x : A, \Gamma \vdash eBody : B}{\Gamma \vdash (\mathsf{Lam}\ x\ A\ eBody) : A \rightarrow B}\ \text{Type-lam} \qquad \frac{\Gamma \vdash e1 : A \rightarrow B \qquad \Gamma \vdash e2 : A}{\Gamma \vdash (\mathsf{App}\ e1\ e2) : B}\ \text{Type-app}$$

$$\frac{\Gamma \vdash e : \mathsf{bool} \qquad \Gamma \vdash eThen : A \qquad \Gamma \vdash eElse : A}{\Gamma \vdash (\mathsf{Ite}\ e\ eThen\ eElse) : A}\ \text{Type-ite}$$

Now we add a rule for Choose:

$$\frac{n \geq 1 \qquad \Gamma \vdash e1 : A \qquad \cdots \qquad \Gamma \vdash en : A}{\Gamma \vdash (\mathsf{Choose}\ e1\ \cdots\ en) : A}\ \text{Type-choose}$$

Similar to Type-ite, Type-choose requires all the expressions to have the same type A because we don't know, statically, which expression will be chosen for evaluation.

[2] Actual technical term. Yes. Really.

Question 5 [30 points]: Demonic Nondeterminism, continued

Q5a [15 points] Complete the following derivation.

$$\dfrac{2 \geq 1 \qquad \dfrac{\dfrac{}{x : \mathsf{num}, \emptyset \vdash (\mathsf{Bfalse}) : \mathsf{bool}}\text{Type-false} \qquad \dfrac{}{x : \mathsf{num}, \emptyset \vdash (\mathsf{Btrue}) : \mathsf{bool}}\text{Type-true}}{x : \mathsf{num}, \emptyset \vdash (\mathsf{Choose}\ (\mathsf{Bfalse})\ (\mathsf{Btrue})) : \mathsf{bool}}\text{Type-choose}}{\emptyset \vdash (\mathsf{Lam}\ x\ \mathsf{num}\ (\mathsf{Choose}\ (\mathsf{Bfalse})\ (\mathsf{Btrue}))) : \mathsf{num} \rightarrow \mathsf{bool}}\text{Type-lam}$$

Q5b [20 points] Implement the above rule by filling in the **type-case** branch for Choose in the typeof function:

```
(define (typeof tc e)   ; typeof : Typing-context E —> (or false Type)
  (type-case E e
    ; ...
    [Ite (e eThen eElse) (and (Tbool? (typeof tc e))
                              (let ([AThen (typeof tc eThen)]
                                    [AElse (typeof tc eElse)])
                                (and (type=? AThen AElse)
                                     AThen)))]
    ; ...
    [Choose (eChoices)
      (if (empty? eChoices)
          #false

          (let ([A (typeof tc (first eChoices))])
            (if (empty? (rest eChoices))
                A
                (and (type=? A (typeof tc (Choose (rest eChoices))))
                     A))))]

      )
    ]))
```