CIS 500 — Software Foundations

Midterm I
Answer key
October 14, 2002

Name:

______________________________________________

Student ID:

______________________________________________
(from your PennCard)

Email

______________________________________________

Status

_____ registered for the course

_____ not registered — just taking the exam for practice

Program

_____ undergrad

_____ undergrad (MSE submatriculant)

_____ CIS MSE

_____ CIS MCIT

_____ CIS PhD

_____ other
Instructions

• This is a closed-book exam: you may not make use of any books or notes.

• You have 80 minutes to answer all of the questions. The entire exam is worth 80 points.

• Questions vary significantly in difficulty, and the point value of a given question is not always exactly proportional to its difficulty. Do not spend too much time on any one question.

• Partial credit will be given. All correct answers are short. The back side of each page may be used as a scratch pad.

• Good luck!
Untyped lambda-calculus

1. (2 points) We have seen that a linear expression like \( \lambda x. \lambda y . x y x \) is shorthand for an abstract syntax tree that can be drawn like this:

\[
\begin{array}{c}
\lambda x \\
\mid \\
\lambda y \\
\mid \\
apply \\
\mid \\
apply \\
\mid \\
x \\
\end{array}
\]

Draw the abstract syntax trees corresponding to the following expressions:

(a) a b c

Answer:

\[
\begin{array}{c}
apply \\
\mid \\
apply \\
\mid \\
c \\
\mid \\
a \\
b \\
\end{array}
\]

(b) (\( \lambda x. b \)) (c d)

Answer:

\[
\begin{array}{c}
apply \\
\mid \\
\lambda x \\
\mid \\
apply \\
\mid \\
b \\
c \\
d \\
\end{array}
\]

Grading scheme: 1 point for each part. No partial credit awarded.

2. (10 points) Write down the normal forms of the following \( \lambda \)-terms:

(a) (\( \lambda t. \lambda f. t \)) (\( \lambda t. \lambda f. f \)) (\( \lambda x. x \))

Answer: \( \lambda t. \lambda f. f \)

(b) (\( \lambda x. x \)) (\( \lambda x. x \)) (\( \lambda x. x \)) (\( \lambda x. x \))

Answer: \( \lambda x. x \)

(c) \( \lambda x. x (\lambda x. x) (\lambda x. x) \)

Answer: \( \lambda x. x (\lambda x. x) (\lambda x. x) \)

(d) (\( \lambda x. x (\lambda x. x) \)) (\( \lambda x. x (\lambda x. x x) \))

Answer: \( \lambda x. x x \)

(e) (\( \lambda x. x x x \)) (\( \lambda x. x x x \))

Answer: No normal form

Grading scheme: Binary. 2 points each.
3. (4 points) Recall the following abbreviations from Chapter 5:

\[ \text{tru} = \lambda t. \lambda f. t \]
\[ \text{fls} = \lambda t. \lambda f. f \]
\[ \text{not} = \lambda b. b \text{ fls } \text{ tru} \]

Complete this definition of a lambda term that takes two church booleans, b and c, and returns the logical "exclusive or" of b and c.

\[ \text{xor} = \lambda b. \lambda c. \ldots \]

Some possible answers:

\[ \text{xor} = \lambda b. \lambda c. b \ (\text{not } c) \ c \]
\[ \text{xor} = \lambda b. \lambda c. b \ (c \text{ fls } \text{ tru}) \ c \]

Grading scheme: Points awarded roughly proportional to the number of correct lines in the XOR truth table.

4. (8 points) A list can be represented in the lambda-calculus by its fold function. (OCaml’s name for this function is fold_right; it is also sometimes called reduce.) For example, the list [x, y, z] becomes a function that takes two arguments c and n and returns c x (c y (c z n))). The definitions of nil and cons for this representation of lists are as follows:

\[ \text{nil} = \lambda c. \lambda n. n; \]
\[ \text{cons} = \lambda h. \lambda t. \lambda c. \lambda n. c\ h\ (t\ c\ n); \]

Suppose we now want to define a \( \lambda \)-term append that, when applied to two lists \( l_1 \) and \( l_2 \), will append \( l_1 \) to \( l_2 \) — i.e., it will return a \( \lambda \)-term representing a list containing all the elements of \( l_1 \) and then those of \( l_2 \). Complete the following definition of append.

\[ \text{append} = \lambda l_1. \lambda l_2. \lambda c. \lambda n. \ldots \]

Answer:

\[ \text{append} = \lambda l_1. \lambda l_2. \lambda c. \lambda n. l_1\ c\ (l_2\ c\ n) \]

Grading scheme: Incorrect recursive definitions of append were awarded partial credit. Points deducted for each incorrect user of cons and nil.
5. (6 points) Recall the call-by-value fixed-point combinator from Chapter 5:

\[ \text{fix} = \lambda f. (\lambda x. f (\lambda y. x \ x \ y)) (\lambda x. f (\lambda y. x \ x \ y)); \]

We can use \text{fix} to write a function \text{sumupto} that, given a Church numerals \( m \), calculates the sum of all the numbers less than or equal to \( m \), as follows.

\[ g = \lambda f. \lambda m. \]
\[ (\text{iszro} \ m) \]
\[ (\lambda x. c_0) \]
\[ (\lambda x. \text{plus} \ _____ \ (_____ (\text{prd} \ m))) \]
\[ \text{tru}; \]
\[ \text{sumupto} = \text{fix} \ g; \]

Fill in the two omitted subterms.

\textbf{Answer:}

\[ g = \lambda f. \lambda m. \]
\[ (\text{iszro} \ m) \]
\[ (\lambda x. c_0) \]
\[ (\lambda x. \text{plus} \ m \ (f (\text{prd} \ m))) \]
\[ \text{tru}; \]

\textit{Grading scheme: First blank is worth 2 points, second blank is worth 4 points (roughly).}
6. (4 points) Suppose we have defined the naming context $\Gamma = a, b, c, d$. What are the deBruijn representations of the following $\lambda$-terms?

(a) $\lambda x. \lambda y. x y d$
   
   Answer: $\lambda. \lambda.102$

(b) $\lambda x. c (\lambda y. (c y) x) d$
   
   Answer: $\lambda.2(\lambda.(30)1)1$

Grading scheme: One point deducted for each incorrect character/set of parens.

7. (4 points) Write down (in deBruijn notation) the terms that result from the following substitutions.

(a) $[0 \rightarrow \lambda.0][(\lambda.01)1]$
   
   Answer: $(\lambda.0(\lambda.0))1$

(b) $[0 \rightarrow \lambda.01][(\lambda.01)0]$
   
   Answer: $(\lambda.0(\lambda.02))(\lambda.01)$

Grading scheme: One point deducted for each incorrect character/set of parens.
Typed arithmetic expressions

The full definition of the language of typed arithmetic and boolean expressions is reproduced, for your reference, on page 11.

8. (9 points) Suppose we add the following new rule to the evaluation relation:

\[
\text{succ } \text{true} \rightarrow \text{pred (succ true)}
\]

Which of the following properties will remain true in the presence of this rule? For each one, write either “remains true” or else “becomes false,” plus (in either case) a one-sentence justification of your answer.

(a) Termination of evaluation (for every term \( t \) there is some normal form \( t' \) such that \( t \rightarrow^* t' \))
   Answer: Becomes false. For example, the term \( \text{succ true} \) has no normal form.

(b) Progress (if \( t \) is well typed, then either \( t \) is a value or else \( t \rightarrow t' \) for some \( t' \))
   Answer: Remains true. Adding a new evaluation rule can only make it easier for the progress property to hold.

(c) Preservation (if \( t \) has type \( T \) and \( t \rightarrow t' \), then \( t' \) also has type \( T \))
   Answer: Remains true: \( \text{succ true} \) is not well typed (nor is any term containing it), so it doesn’t matter what it evaluates to.

Grading scheme: -3 for each wrong answer. One point awarded for correct answer. One point awarded for partial explanation (given generously). One point awarded for complete explanation (given sparingly).

9. (9 points) Suppose, instead, that we add this new rule to the evaluation relation:

\[
t \rightarrow \text{if } \text{true then } t \text{ else succ false}
\]

Which of the following properties remains true? (Answer in the same style as the previous question.)

(a) Termination of evaluation (for every term \( t \) there is some normal form \( t' \) such that \( t \rightarrow^* t' \))
   Answer: Becomes false. For any term \( t \), we can evaluate \( t \rightarrow \text{if true then } t \text{ else succ false} \rightarrow t \rightarrow \ldots \)

(b) Progress (if \( t \) is well typed, then either \( t \) is a value or else \( t \rightarrow t' \) for some \( t' \))
   Answer: Remains true. As above, adding a new evaluation rule can only make it easier for the progress property to hold.

(c) Preservation (if \( t \) has type \( T \) and \( t \rightarrow t' \), then \( t' \) also has type \( T \))
   Answer: Becomes false: a well typed term like \( \text{zero} \) can now evaluate to the ill-typed term \( \text{if true then zero else succ false} \).

Grading scheme: -3 for each wrong answer. One point awarded for correct answer. One point awarded for partial explanation (given generously). One point awarded for complete explanation (given sparingly).
10. (9 points) Suppose, instead, that we add a new type, Funny, and add this new rule to the typing relation:

\[
\text{if } \text{true} \text{ then false else false : Funny}
\]

Which of the following properties remains true? (Answer in the same style as the previous question.)

(a) Termination of evaluation (for every term \( t \) there is some normal form \( t' \) such that \( t \rightarrow^* t' \))
   
   \text{Answer: Remains true. Adding typing rules doesn’t change the evaluation relation or its properties.}

(b) Progress (if \( t \) is well typed, then either \( t \) is a value or else \( t \rightarrow t' \) for some \( t' \))
   
   \text{Answer: Remains true. This rule doesn’t make any new terms well typed.}

(c) Preservation (if \( t \) has type \( T \) and \( t \rightarrow t' \), then \( t' \) also has type \( T \))
   
   \text{Answer: Becomes false: for example, the term if true then false else false has type Funny, but reduces to false, which does not have type Funny.}

\text{Grading scheme: -3 for each wrong answer. One point awarded for correct answer. One point awarded for partial explanation (given generously). One point awarded for complete explanation (given sparingly).}
**Simply typed lambda-calculus**

The definition of the simply typed lambda-calculus with booleans is reproduced for your reference on page 13.

11. (6 points) Write down the types of each of the following terms (or “ill typed” if the term has no type).

(a) \( \lambda x: \text{Bool}. \, x \ x \)
   
   *Answer: ill typed*

(b) \( \lambda f: \text{Bool} \to \text{Bool}. \, \lambda g: \text{Bool} \to \text{Bool}. \, g \ (f \ (g \ \text{true})) \)

   *Answer: \((\text{Bool} \to \text{Bool}) \to (\text{Bool} \to \text{Bool}) \to \text{Bool}\)*

(c) \( \lambda h: \text{Bool}. \, (\lambda i: \text{Bool} \to \text{Bool}. \ i \ \text{false}) \ (\lambda k: \text{Bool}. \ \text{true}) \)

   *Answer: \(\text{Bool} \to \text{Bool}\)*

*Grading scheme: Binary. Partial credit awarded for very, very close answers (like misplaced parens).*
12. (9 points) Recall the rules for “big-step evaluation” of arithmetic and boolean expressions from HW 3.

\[
\begin{align*}
v & \Downarrow v \\
\text{if } t_1 \text{ then } t_2 \text{ else } t_3 & \Downarrow v_2 \\
\text{if } t_1 \text{ then } t_2 \text{ else } t_3 & \Downarrow v_3 \\
succ t_1 & \Downarrow succ \, v_1 \\
\text{succ } t_1 & \Downarrow succ \, v_1 \\
\text{pred } t_1 & \Downarrow 0 \\
\text{pred } t_1 & \Downarrow 0 \\
\text{iszero } t_1 & \Downarrow \text{true} \\
\text{iszero } t_1 & \Downarrow \text{false} \\
\end{align*}
\]

The following OCaml definitions implement this evaluation relation *almost correctly*, but there are three mistakes in the eval function—one each in the TmIf, TmSucc, and TmPred cases of the outer match. Show how to change the code to repair these mistakes. (Hint: all the mistakes are *omissions*.)

```ocaml
let rec isnumericval t = match t with
  | TmZero(_) -> true
  | TmSucc(_,t1) -> isnumericval t1
  | _ -> false

let rec isval t = match t with
  | TmTrue(_) -> true
  | TmFalse(_) -> true
  | t when isnumericval t -> true
  | _ -> false

let rec eval t = match t with
  | v when isval v -> v
  | TmIf(_,t1,t2,t3) ->
    (match t1 with
     | TmTrue _ -> eval t2
     | TmFalse _ -> eval t3
     | _ -> raise NoRuleApplies)
  | TmSucc(fi,t1) ->
    (match eval t1 with
     | nv1 -> TmSucc(dummyinfo, nv1)
     | _ -> raise NoRuleApplies)
  | TmPred(fi,t1) ->
    (match eval t1 with
     | TmZero _ -> TmZero(dummyinfo)
     | _ -> raise NoRuleApplies)
  | TmIsZero(fi,t1) ->
    (match eval t1 with
     | TmZero _ -> TmTrue(dummyinfo)
     | TmSucc(_, _) -> TmFalse(dummyinfo)
     | _ -> raise NoRuleApplies)
  | _ -> raise NoRuleApplies
```

Answer:

- In the TmIf clause, match t1 with should be match (eval t1) with.
- In the TmSucc clause, the guard nv1 → ... should be
  \[ nv1 \text{ when isnumericval} \] or, equivalently, the body of the clause,
  \[ \text{TmSucc (dummyinfo, } \text{nv1), should be replaced by} \]
  \[ \text{if isnumericval } \text{nv1 then TmSucc (dummyinfo, } \text{nv1} \text{) else raise NoRuleApplies} \]
- In the TmPred clause, the whole case
  \[ | \text{TmSucc(_, } \text{nv1) } \text{→ } \text{nv1} \]
  is missing from the inner match (it should follow the TmZero case).

Grading scheme: 3 points for each bug. 1 point for finding the correct location of agv. 2 points for correct fix. 1 point for flawed fix. 0 for fixing "wrong" bug. No penalty for redundant call to isnumericval.
For reference: Untyped boolean and arithmetic expressions

Syntax

\( t ::= \)

- true
- false
- if \( t \) then \( t \) else \( t \)
- 0
- succ \( t \)
- pred \( t \)
- iszero \( t \)

\( v ::= \)

- true
- false
- \( n v \)

\( n v ::= \)

- 0
- succ \( n v \)

\( T ::= \)

- Bool
- Nat

Evaluation

\[
\begin{align*}
\text{if true then } t_2 \text{ else } t_3 & \rightarrow t_2 \\
\text{if false then } t_2 \text{ else } t_3 & \rightarrow t_3 \\
(\text{E-If}) \\
\text{if } t_1 \text{ then } t_2 \text{ else } t_3 & \rightarrow \text{if } t'_1 \text{ then } t_2 \text{ else } t_3 \\
(\text{E-If}) \\
\text{succ } t_1 & \rightarrow \text{succ } t'_1 \\
(\text{E-Succ}) \\
\text{pred } 0 & \rightarrow 0 \\
(\text{E-PredZero}) \\
\text{pred } (\text{succ } n v_1) & \rightarrow n v_1 \\
(\text{E-PredSucc}) \\
\text{pred } t_1 & \rightarrow \text{pred } t'_1 \\
(\text{E-Pred}) \\
\text{iszero } 0 & \rightarrow \text{true} \\
(\text{E-IszeroZero}) \\
\text{iszero } (\text{succ } n v_1) & \rightarrow \text{false} \\
(\text{E-IszeroSucc}) \\
\text{iszero } t_1 & \rightarrow \text{iszero } t'_1 \\
(\text{E-Iszero})
\end{align*}
\]

continued on next page...
Typing

true : Bool  \quad \text{(T-TRUE)}
false : Bool  \quad \text{(T-FALSE)}

\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T  \quad \text{(T-IF)}

0 : \text{Nat}  \quad \text{(T-ZERO)}

\begin{array}{c}
\text{succ } t_1 : \text{Nat}  \\
\text{pred } t_1 : \text{Nat}
\end{array}

\text{iszero } t_1 : \text{Bool}  \quad \text{(T-ISZERO)}
For reference: Simply typed lambda calculus with booleans

Syntax

\[ t ::= \]

- `true`
- `false`
- `if t then t else t`
- `x`
- `\lambda x:T.t`
- `tt`

\[ v ::= \]

- `true`
- `false`
- `\lambda x:T.t`

\[ T ::= \]

- `Bool`
- `T \rightarrow T`

Evaluation

\[
\frac{t \rightarrow t'}{if \; t \; then \; t_2 \; else \; t_3 \rightarrow t_2} \quad (E-IFTRUE)
\]

\[
\frac{t \rightarrow t'}{if \; false \; then \; t_2 \; else \; t_3 \rightarrow t_3} \quad (E-IFFALSE)
\]

\[
\frac{t_1 \rightarrow t_1'}{if \; t_1 \; then \; t_2 \; else \; t_3 \rightarrow if \; t_1' \; then \; t_2 \; else \; t_3} \quad (E-IF)
\]

\[
\frac{t_1 \rightarrow t_1'}{t_1 \; t_2 \rightarrow t_1' \; t_2} \quad (E-APP1)
\]

\[
\frac{t_2 \rightarrow t_2'}{t_2 \rightarrow t_2'} \quad (E-APP2)
\]

\[
\frac{}{(\lambda x:T_1.t_2) \; v_2 \rightarrow [x \mapsto v_2] \; t_2} \quad (E-APPABS)
\]

Typing

\[
true : Bool \quad (T-TRUE)
\]

\[
false : Bool \quad (T-FALSE)
\]

\[
t_1 : Bool \quad t_2 : T \quad t_3 : T \quad (T-IF)
\]

\[
\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad (T-VAR)
\]

\[
\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1. t_2 : T_1 \rightarrow T_2} \quad (T-ABS)
\]

\[
\frac{\Gamma \vdash t_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1 \; t_2 : T_12} \quad (T-APP)
\]