CS 421 Spring 2013 Final Exam

Tuesday, May 7, 2013

Name	
NetID	

- You have **180 minutes** to complete this exam
- This is a **closed book** exam.
- Do not share anything with other students. Do not talk to other students. Do not look at another student's exam. Do not expose your exam to easy viewing by other students. Violation of any of these rules will count as cheating.
- If you believe there is an error, or an ambiguous question, seek clarification from one of the TAs. You must use a whisper, or write your question out.
- Including this cover sheet, there are 14 pages to the exam. Please verify that you have all 14 pages.
- Please write your name and NetID in the spaces above, and at the top of every page.

Question	**Value**	**Score**
1	10	
2	5	
3	5	
4	15	
5	15	
6	12	
7	10	
8	8	
9	5	
10	5	
11	10	
12	5 (ec)	
13	5 (ec)	
Total	**100**	
EC	**10**	

1. (10 pts) Match each term on the left with the most appropriate phrase on the right:

__e__ Token

__j__ Machine-independent optimization

__b__ Machine-dependent optimization

__c__ Nullable non-terminal

__i__ Extended context-free grammar

__a__ Type error

__f__ V-table

__d__ Higher-order function

__g__ Type-inference

__h__ Type scheme

(a) Can be found at compile time

(b) Transformation of IR to native code

(c) Can produce empty string

(d) Function that takes a function as argument or returns a function a function as result

(e) Unit of input text meaningful for parsing

(f) Data structure for implementing inheritance

(g) Algorithm to fill in type declarations

(h) Type with some type variables quantified

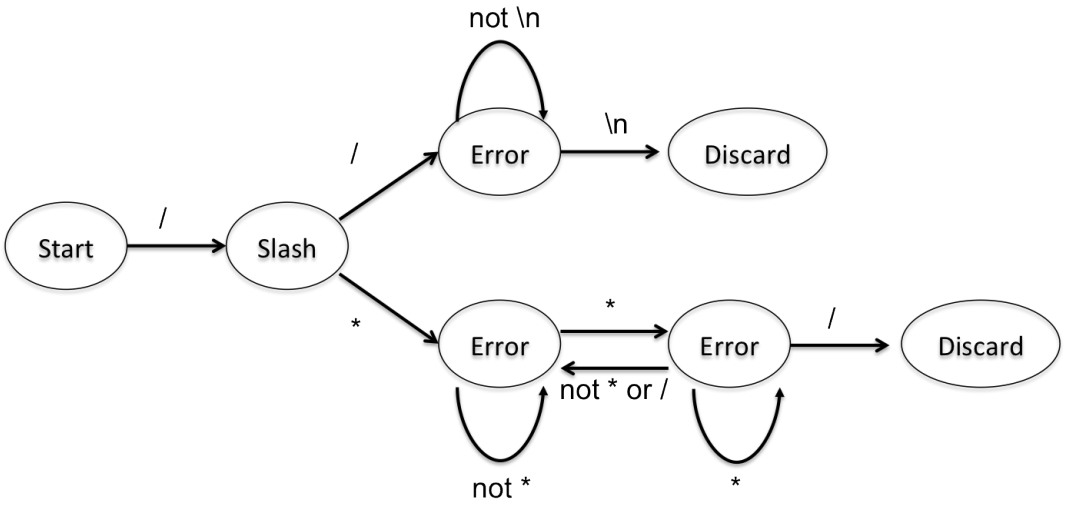
(i) Contains regular expressions in productions

(j) Beneficial transformation of IR

2. (5 points) In this problem, we use the following types of comments:

   - C++: Starts with //, ends with newline
   - C: Starts with /∗, ends with ∗/, no nesting

   (a) (3 points) Draw a single DFA for C++ and C comments. As we did in class, the states should be labelled either Start, Error, Discard, or one of the token types (Slash for a single /). Note that we have an error if we start a comment and do not end it.



   (b) (2 points) Give a regular expression for C++ comments.

```
'/' '/' [^'\n']*
```

3. (5 points) Consider the OCaml type `token` of tokens representing integers, floats, and operators + and -:

```
type token = PLUS | MINUS | INT of int | FLOAT of float
```

A float is at least one digit optionally followed by a decimal point followed by zero or more digits, optionally followed by `E` or `e` followed by at least one digit (there is no sign before the exponent). (Examples: `1.0e2`, `2.`, `10e5`, and `02.3E0` are floats, but `.23`, `e8`, `9E`, and `9e+3` are not.)

Give an ocamllex specification converting an input stream to a list of tokens; characters not matching an integer, float, '+' or '-' are ignored. You can use the following functions:

- `int_of_string : string -> int`
- `float_of_string : string -> float`

(Note that `float_of_string` accepts strings in the format described above, so you simply need to capture the characters and pass them to that function.)

Hint: you will need to use "`as`" patterns: *`regexp`* `as` *`name`* matches the regular expression and assigns the matched string to *name* (which can then be used in the action).

Complete the following:

```
let digit = ['0' - '9']

rule tokenize = parse
    (* add your rules below *)
      digit+ as num    { INT (int_of_string num) }
    | ( digit+ ('.' digit*)? (['E''e'] digit+)? ) as num
                                { FLOAT (float_of_string num) }
    | '+'              { PLUS }
    | '-'              { MINUS }
    | _                { tokenize lexbuf }
```

One alternative for float rule (among many) is three rules:

```
    | digit+ '.' digit* as num  { FLOAT (float_of_string num) }
    | digit+ ['E''e'] digit+ as num  { FLOAT (float_of_string num) }
    | digit+ '.' digit* ['E''e'] digit+ as num  { FLOAT (float_of_string num) }
```

4. (15 pts) This multi-part question concerns expression grammars. Consider this grammar:

$$\mathcal{G}_0\text{: } E \to E \texttt{ + } E \mid E \texttt{ - } E \mid E \texttt{ * } E \mid E \texttt{ / } E \mid \mathit{ident} \mid \texttt{(} E \texttt{)}$$

(a) (4 pts) Here is a stratified grammar for the same language:

$\mathcal{G}_1$: $E \to T \mid T \texttt{ + } E \mid T \texttt{ - } E$
$T \to P \mid P \texttt{ * } T \mid P \texttt{ / } T$
$P \to \mathit{ident} \mid \texttt{(} E \texttt{)}$

Answer these multiple choice questions concerning $\mathcal{G}_1$:

i. __b__ $\mathcal{G}_1$ is (a) ambiguous, (b) unambiguous, or (c) impossible to tell from provided data.

ii. __c__ $\mathcal{G}_1$ gives (a) multiplication precedence over division, (b) division precedence over multiplication, (c) multiplication and division the same precedence, or (d) the question of precedence between multiplication and division is meaningless.

iii. __b__ In $\mathcal{G}_1$, addition and subtraction (a) are left-associative, (b) are right-associative, (c) do not have any associativity enforced by the grammar.

iv. __b__ $\mathcal{G}_1$ is (a) LL(1), or (b) not LL(1).

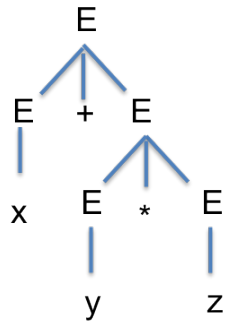
(b) (4 pts) Perform left-factoring on $\mathcal{G}_1$. (Hint: the resulting grammar has exactly 10 productions.)

$E \to T\ E'$
$E' \to \epsilon \mid \texttt{ + } E \mid \texttt{ - } E$
$T \to P\ T'$
$T' \to \epsilon \mid \texttt{ * } T \mid \texttt{ / } T$
$P \to \mathit{ident} \mid \texttt{(} E \texttt{)}$

(c) (4 pts) Consider again the original grammar:

$$\mathcal{G}_0\text{: } E \rightarrow E \texttt{ + } E \mid E \texttt{ - } E \mid E \texttt{ * } E \mid E \texttt{ / } E \mid \mathit{ident} \mid \texttt{(} E \texttt{)}$$

Here is a parse tree for `x + y * z`:



Give the shift-reduce parse corresponding to this parse tree. It has exactly 11 steps. For reduce actions, be sure to give the production being used.

Action	**Stack**	**Input**
Shift		`x + y * z`
R $E \rightarrow id$	x	`+y*z`
Sh	E	`+y*z`
Sh	E+	`y*z`
R $E \rightarrow id$	E+y	`*z`
Sh	E+E	`*z`
Sh	E+E*	`z`
R $E \rightarrow id$	E+E*z	eof
R $E \rightarrow E * E$	E+E*E	eof
R $E \rightarrow E + E$	E+E	eof
Accept	E	eof

(d) (3 pts) Give ocamlyacc precedence declarations enforcing the usual precedence and associativities of +, -, *, and /. (Use token names `PLUS`, `MINUS`, `STAR`, and `SLASH`.)

```
%left PLUS MINUS
%left STAR SLASH
```

5. (15 pts) In this question, we will ask you to fill in SOS rules in the style of MP6 (MiniJava interpretation with one-level store), MP7 (two-level store), and MP8 (compilation). As a reminder, the judgments in these three systems are:

	Expressions	**Statements**
One-level store (MP6)	$e, \sigma, \pi \Downarrow v$	$S, \sigma, \pi \Rightarrow \sigma'$
Two-level store (MP7)	$e, (\rho, \eta), \pi \Downarrow v, \eta'$	$S, (\rho, \eta), \pi \Rightarrow \rho', \eta'$
Compilation (MP8)	$e, loc \rightsquigarrow code$	$S, m \rightsquigarrow code, m'$

Specifically, for each of the three assignments, we will ask you to give rules for `whilebreak`. For interpretation (MP6 and 7), there are two rules, one for when the condition is true and one for when it is false. For compilation, there is just one rule (and it is not recursive). In all three parts of this problem, you need to fill in the blank lines.

The statement `whilebreak` S_1 `(`*`cond`*`)` S_2 executes S_1, and then tests the condition; if the condition is FALSE, it executes S_2 and S_1 and tests the condition again; repeat until the condition is TRUE.

(a) (5 pts) One-level store:

$$\frac{S_1, \sigma, \pi \Rightarrow \sigma' \qquad e, \sigma', \pi \Downarrow true \qquad S_2, \sigma', \pi \Rightarrow \sigma''}{\texttt{whilebreak } S_1 \texttt{ (}e\texttt{) } S_2, \sigma, \pi \Rightarrow \sigma'''}$$

$$\frac{S_1, \sigma, \pi \Rightarrow \sigma' \qquad e, \sigma, \pi \Downarrow false}{\texttt{whilebreak } S_1 \texttt{ (}e\texttt{) } S_2, \sigma, \pi \Rightarrow \sigma'}$$

(b) (5 pts) Two-level store:

$$\frac{S_1, (\rho, \eta), \pi \Rightarrow \rho', \eta' \qquad e, (\rho', \eta'), \pi \Downarrow true, \eta'' \qquad S_2, (\rho', \eta''), \pi \Rightarrow \rho'', \eta'''}{\texttt{whilebreak } S_1 \texttt{ (}e\texttt{) } S_2, (\rho, \eta), \pi \Rightarrow \hat{\rho}, \hat{\eta}}$$

where (ρ'', η''') fills the blank.

$$\frac{S_1, (\rho, \eta), \pi \Rightarrow \rho', \eta' \qquad S_1, (\rho', \eta'), \pi \Downarrow false, \eta''}{\texttt{whilebreak } S_1 \texttt{ (}e\texttt{) } S_2, (\rho, \eta), \pi \Rightarrow \eta''}$$

(c) (5 pts) Compilation

`whilebreak` S_1 `(`e`)` S_2, $m \rightsquigarrow$ il1 @ il @ [CJUMP loc, $m' + |\text{il}| + 1$, $m'' + 1$]

@ il2 @ [JUMP m], $m'' + 1$

S_1, $m \rightsquigarrow$ il1, m'

e, loc $\rightsquigarrow$ il

S_2, $m' + |\text{il}| + 1 \rightsquigarrow$ il2, m''

6. (12 pts) In the various operational semantics for OCaml, the most interesting rule in most cases was application.

(a) (6 pts) Fill in the application rule for those cases:

Substitution ($e \Downarrow v$):

(App) $e_1\ e_2 \Downarrow v$
$e_1 \Downarrow$ `fun` x `->` e
$e_2 \Downarrow v'$
$e[v'/x] \Downarrow v$

Environment ($e, \rho \Downarrow v$):

(App) $e_1\ e_2, \rho \Downarrow v$
$e_1, \rho \Downarrow <$`fun` x `->` $e, \rho' >$
$e_2, \rho \Downarrow v'$
$e, \rho'[x \mapsto v'] \Downarrow v$

Lazy ($e \Downarrow_\ell v$):

(App) $e_1\ e_2 \Downarrow_\ell v$
$e_1 \Downarrow_\ell$ `fun` x `->` e
$e[e_2/x] \Downarrow_\ell v$

(b) (6 pts) In lecture 24, we introduced the side-effecting operations of OCaml. To give their operational semantics, we start with the environment model, but we need a two-level store (just as for Java). The new judgments have the form

$$e,\ (\rho,\ \omega) \Downarrow v,\ \omega'$$

where ω is the "heap." Here are two of the rules:

(Fun) `fun` x `->` $e,\ (\rho,\ \omega) \Downarrow <$`fun` x `->` $e,\ \rho >,\ \omega$

(Ref) $$\frac{\texttt{ref}\ e,\ (\rho,\ \omega) \Downarrow \ell,\ \omega'[\ell \mapsto v]}{e,\ (\rho,\ \omega) \Downarrow v,\ \omega'} \quad \text{(where } \ell \text{ is a fresh location not used in } \omega')$$

Give the rules for the following expressions. Recall that the dereferencing operator, `!`, takes an expression that evaluates to a heap location and returns its contents; its type is α `ref` $\rightarrow \alpha$. The assignment operator, `:=`, has a left-hand argument that evaluates to a location and a right-hand argument that evaluates to some value, and it stores the right-hand value into the left-hand location; its type is α `ref` * $\alpha \rightarrow$ unit. The addition operator evaluates its arguments left to right.

(Plus) $$\frac{e_1,\ (\rho,\omega) \Downarrow i_1,\ \omega' \qquad e_2,\ (\rho,\omega') \Downarrow i_2,\ \omega''}{e_1 + e_2,\ (\rho,\ \omega) \Downarrow i_1 + i_2,\ \omega''}$$

(Dereference) $$\frac{e,\ (\rho,\omega) \Downarrow \ell,\ \omega' \qquad \text{(where } \omega'(\ell) = v\text{)}}{\texttt{!}\ e,\ (\rho,\ \omega) \Downarrow v,\ \omega'}$$

(Assign) $$\frac{e_1,\ (\rho,\omega),\ \Downarrow,\ \ell,\ \omega' \qquad e_2,\ (\rho,\omega'),\ \Downarrow,\ v,\ \omega''}{e_1 := e_2,\ (\rho,\ \omega) \Downarrow (),\ \omega''[\ell \mapsto v]}$$

7. (10 pts) Recall the specification of `fold_right`:

$$\texttt{fold\_right}\ f\ [x_1; x_2; \ldots; x_n]\ z = f\, x_1\, (f\, x_2\, (\ldots\ f\, x_n\, z) \ldots))$$

(which is just z if $n = 0$).

(a) (2 pts) Give a recursive definition for `fold_right`:

```
let rec fold_right f lis z = if lis=[] then z
                   else f (hd lis) (fold_right f (tl lis) z)
```

or

```
let rec fold_right f lis z = match lis with
       [] -> z
     | h::t -> f h (fold_right f t z)
```

(b) (8 pts) Define the following functions as "one-liners" by using `fold_right` and an anonymous function:

i. `let sum lis = (* add elements of integer list lis *)`

```
fold_right (+) lis 0    or    fold_right (fun x y -> x+y) lis 0
```

ii. `let big_and lis = (* true iff all elements of bool list lis are true *)`

```
fold_right (&) lis true   or    fold_right (fun x y -> x&y) lis true
```

iii. `let big_or lis = (* true iff at least one element of bool list lis is true *)`

```
fold_right (or) lis false   or    fold_right (fun x y -> x or y) lis false
```

iv. `let map f lis = (* usual definition of map *)`

```
fold_right (fun h t -> f h :: t) lis []
```

8. (8 pts) This question is about creating comparison functions by defining "comparison combinators." A comparison function, or "comparator," is simply a function of type $\alpha \to \alpha \to$ bool:

```
type 'a comparator = 'a -> 'a -> bool
```

Here are a couple of examples; `lt` is an int comparator; `sumcomp` is a function from an int comparator to an (int * int) comparator:

```
let lt = fun x y -> x <= y
let sumcomp = fun comp -> fun (x1,y1) (x2,y2) -> comp (x1+y1) (x2+y2)
```

Define these functions (2 pts each):

(a) `invert: 'a comparator -> 'a comparator` inverts its argument's arguments: `(invert lt) 3 4` = false; `(invert lt) 4 3` = true; `(invert lt) 3 3` = true.

```
let invert comp = fun x y -> comp y x
```

(b) `first: 'a comparator -> ('a*'b) comparator` creates a comparison function that uses just the first element of a pair: `(first lt) (3, 4.3) (4, 2.1)` = true; `(first lt) (4, 4.3) (3, 8.1)` = false.

```
let first comp = fun (x1,x2) (y1,y2) -> comp x1 y1
```

(c) `both: 'a comparator -> ('a*'a) comparator` creates a comparison function that compares both elements of a pair: `(both lt) (3, 4) (4, 5)` = true; `(both lt) (3, 4) (4, 3)` = false.

```
let both comp = fun (x1,x2) (y1,y2) -> comp x1 y1 & comp x2 y2
```

(d) `nthcomp: int -> 'a comparator -> ('a list) comparator` creates a comparison function that compares lists by comparing one of their elements: `nthcomp 1 lt [1;2;3;4] [1;3;5;7]` = true; `nthcomp 3 lt [1;2;3;4] [1;3;5;3]` = false. (You may use function `nth: 'a list -> int -> 'a`.)

```
let nthcomp n comp = fun lis1 lis2 -> comp (nth lis1 n) (nth lis2 n)
```

9. (5 pts) In the following, fill in `V` if the given term violates the value restriction, `NV` if it does not violate it.

   NV ___ `let f = List.map (fun x -> x + 2);;`

   V ___ `let f = List.map (fun x -> x);;`

   NV ___ `let f = fun lis -> List.map (fun x -> x) lis;;`

   NV ___ `let f = ref (fun x -> x + 2);;`

   NV ___ `let f = ref 3;;`

10. (5 pts) In the following, enter `T` if the type on the left is an instance of the type scheme on the right, `F` if it is not.

    T ___ int→int $\leq \forall\alpha.\alpha \to \alpha$

    T ___ int→int $\leq \forall\alpha.$int $\to \alpha$

    F ___ int→ $\beta \leq \forall\alpha.\alpha \to \gamma$

    F ___ float→ $(\beta \to$int$) \leq \forall\beta.\beta \to (\beta \to$int$)$

    T ___ float→(int→int) $\leq \forall\alpha, \beta.$float→ $(\alpha \to \beta)$

11. (10 points) Using the rules on the last page of this exam, type check the following. We have given underlines for each judgment at the correct indentation level.

$\emptyset \vdash$ `let id:(alpha->alpha) = fun x:alpha -> x in id[bool->bool] true` : bool

$\emptyset \vdash$ `fun x:alpha -> x :  alpha -> alpha`

{ x : alpha } $\vdash$ x : alpha

{ id: $\forall$ alpha. alpha $\rightarrow$ alpha } $\vdash$ `id[bool->bool] true :  bool`

{ id: $\forall$ alpha. alpha $\rightarrow$ alpha } $\vdash$ `id[bool->bool] :  bool->bool`

{ id: $\forall$ alpha. alpha $\rightarrow$ alpha } $\vdash$ `true :  bool`

12. (5 pts extra credit) This is a variant of a question asked on midterm 1. Given this grammar:

$E \rightarrow$ `let` D `in` E | F
$F \rightarrow G\ F'$
$F' \rightarrow$ `+` F | ϵ
$G \rightarrow$ `int` | `string` | `(` E `)`
$D \rightarrow$ `string =` E

fill in the missing parts of a top-down recognizer for the grammar. You can assume `parseE`, `parseF'`, and `parseD` are already done; you have to write `parseF` and `parseG`. Recall that each of the functions has type `token list` $\rightarrow$ `token list`. You should write `raise SyntaxError` when you detect a syntax error. Here is the list of tokens:

```
type token = LetKW | InKW | Plus | LParen | RParen | Eq
           | Int of int | Id of string | EOF

let rec parseE toklis = (* assume done *)

and parseF toklis = parseF' (parseG toklis)


and parseF' toklis = (* assume done *)

and parseG toklis = match toklis with

     Int i :: toklis' -> toklis'

   | Id s :: toklis' -> toklis'

   | LParen :: toklis' -> let toklis'' = parseE toklis'
            in (match toklis'' with
                   RParen :: toklis''' -> toklis'''
                 | _ -> raise SyntaxError)

   | _ -> raise SyntaxError


and parseD toklis = (* assume done *)
```

13. (5 pts extra credit) This question is a variation on a question from midterm 2. Fill in the blanks.

    (a) The three stages of the back-end of a compiler are, in order:

        i. translation from AST to intermediate representation (IR)

        ii. optimization (transformation of IR)

        iii. codegen (translation of IR to native, or machine, code)

    (b) A major disadvantage of reference counting for automatic memory management is its inability to handle cyclic data structures.

    (c) Mark-sweep garbage collection takes time proportional to the size of the heap, while stop-and-copy garbage collection takes time proportional to the size of the reachable data.

    (d) Normally, the statements "`x = f(0);`" and "`{ x = f(0); x = f(0); }`" are equivalent. However, they may give different results if `f` has side effects.

    (e) If `A` is declared in C as `int[10][5]`, and integers are four bytes, then `A[i][j]` is at location address(A) + $i * 20 + j * 4$.

    (f) Programs can be verified in a proof system whose judgments, called Hoare triples, have the form $P\{A\}Q$. The truth of a Hoare triple does not mean that the statement terminates; if we can prove that a statement satisfies a Hoare triple *and* always terminates, we have proven its "total correctness".

Type-checking rules, where Γ is a map from variables to type schemes. τ, τ', τ'' are types. (Feel free to tear off this page.)

(Const) $\Gamma \vdash \text{Int i} : \text{int}$

(Var) $\Gamma \vdash a : \Gamma(a)$
($\Gamma(a)$ a type)

(Fun) $\Gamma \vdash \text{fun } a{:}\tau \text{ -> } e : \tau \to \tau'$
$\Gamma[a{:}\tau] \vdash e : \tau'$

(δ) $\Gamma \vdash e \oplus e' : \tau''$
$\Gamma \vdash e : \tau$
$\Gamma \vdash e' : \tau'$

(App) $\Gamma \vdash e\ e' : \tau'$
$\Gamma \vdash e : \tau \to \tau'$
$\Gamma \vdash e' : \tau$

(True) $\Gamma \vdash \text{true} : \text{bool}$

(False) $\Gamma \vdash \text{false} : \text{bool}$

(PolyVar) $\Gamma \vdash a[\tau] : \tau$
where $\tau \leq \Gamma(a)$
($\Gamma(a)$ a type scheme)

(Let) $\Gamma \vdash \text{let } a{:}\tau \texttt{ = } e \texttt{ in } e' : \tau'$
$\Gamma \vdash e : \tau$
$\Gamma[a{:}\text{GEN}_\Gamma(\tau)] \vdash e' : \tau'$