CS 421 Spring 2012 Practice Final Exam, Part 1

May 4, 2012

This practice exam contains questions similar to those that will appear on the final exam. There are several differences:

- Because there is no set time limit on a practice exam, the exam is longer (note that this is just the first part).
- On the actual exam, many questions will be partially filled in; this is both to reduce the amount of writing you need to do and to make the exam easier to grade. We have done that only minimally on the practice exam.

On the other hand, the practice exam offers a good idea of both the areas that will be covered on the exam and the difficulty level of the questions. Some of the questions also contain comments describing what you should study; if you can do the problems on the practice exam, you should do well on the final, but of course you still need to study the class notes, midterms, and homeworks, since the practice exam does not test everything we covered in class.

1. (*Writing recursive functions on lists and trees, and determining their types (by inspection).*)

   Write a function distance_map that takes a pair of integers and a list of integer pairs, and returns a list of integers which contains the *Manhattan* distance of each point in the list from the given point. Note that Manhattan distance between (x1,y1) and (x2,y2) is $|x1 - x2| + |y1 - y2|$.

```
# let distance_map ... = ...;;
val distance_map: int * int -> (int * int) list -> int list = <fun>
# distance_map (5,5) [(1,2);(3,4);(5,6);(8,7)];;
- : int list = [7;3;1;5]
```

```
let rec distance_map (x0,y0) lis =
  match lis with
      [] -> []
    | (x,y)::t -> (abs(x-x0)+abs(y-y0))
                  :: distance_map t
```

2. (*Functions on abstract syntax trees.*) *Static single assignment form* is a representation of statements in which no variable is assigned to in two different statements. An ordinary sequence of statements can be transformed into SSA form by renaming variables. For example, the sequence of assignments on the right calculates the same values as the one on the left:

```
x = 3;                    x1 = 3;
y = x + 7;                y1 = x1 + 7;
x = y + x;                x2 = y1 + x1;
z = x + 1;                z1 = x2 + 1;
```

So the idea of tansforming to SSA form is just to iterate over the statements, remembering the current name of each variable, and then change the name each time the variable is redefined (i.e. appears on the left-hand side of an assignment). Things actually get a little complicated when there are if statements and loops, but in this problem we will just transform sequences of assignment statements to SSA form. Specifically, given types

```
type exp = Int of int | Var of string | Add of exp * exp
type asgn = Asgn of string * exp
```

we will define `ssa: asgn list → asgn list` that will transform its argument into SSA form. For example,

```
let stmts = [Asgn("x", Int 3);
             Asgn("y", Add(Var "x", Int 7));
             Asgn("x", Add(Var "y", Var "x"));
             Asgn("z", Add(Var "x", Int 1))];;
ssa stmts;;
[Asgn ("x1", Int 3);
 Asgn ("y1", Add (Var "x1", Int 7));
 Asgn ("x2", Add (Var "y1", Var "x1"));
 Asgn ("z1", Add (Var "x2", Int 1))]
```

We will do this in two stages. We actually want to define an auxiliary function

```
let rec ssa' (sl:asgn list) (m:varmap) : asgn list =
  ...

let ssa sl = ssa' sl emptyvarmap
```

So we will start by defining the type `varmap`, which maps variable names to integers, representing the current version of the variable name.

```
type varmap = (string * int) list
let emptyvarmap = []
```

Define the following functions recursively:

```
(* update increments the integer associated with x by 1.
   If x is not in the map, it is added with integer 1. *)
let rec update (x:string) (m:varmap) : varmap = match m with
    [] -> [(x,1)]
  | (y,n)::t -> if y=x then (y,n+1)::t else (y,n)::update x t

(* fetch returns the string consisting of x concatenated with
   the integer associated with x in m; if x is not in m,
   return x concatenated with "0" *)
let rec fetch (x:string) (m:varmap) : string = match m with
    [] -> x ^ "0"
  | (y,n)::t -> if y=x then x^(string_of_int n)
                else fetch x t

(* rename renames every variable in e with the name obtained from
   m (i.e. each variable x becomes fetch x m, which is x concatenated
   with an integer) *)
let rec rename (e:exp) (m:varmap) =
  match e with
    Int i -> Int i
  | Var x -> fetch x m
  | Plus(e1,e2) -> Plus(rename e1 m,
                        rename e2 m)
```

We are now in a position to write the function `ssa'`. The idea is simply to rewrite the first assignment in the list according to the map, and then recursively transform the remaining assignments, after updating the map:

```
let rec ssa' (sl:asgn list) (m:varmap) : asgn list =
  match sl with
    [] -> []
  | (Asgn(x,e))::tsl ->
      let e' = rename e m
      and m' = update x m
      in Asgn(fetch x m', e')
         :: ssa' tsl m'
```

3. (*Lexing using regular expressions and DFAs.*)

An `XML node` — or rather, a simplified version of an XML node — starts with `<`, then has a tag, and then zero or more attributes, followed by either `>` or `/>`. A tag consists of one or more lower-case letters. An attribute is a single space followed by a tag, an equal sign, and a string (double quotes containing any characters at all except double quotes). We are assuming that the spaces at the start of each attribute are the only spaces allowed. For example, here is an XML node:

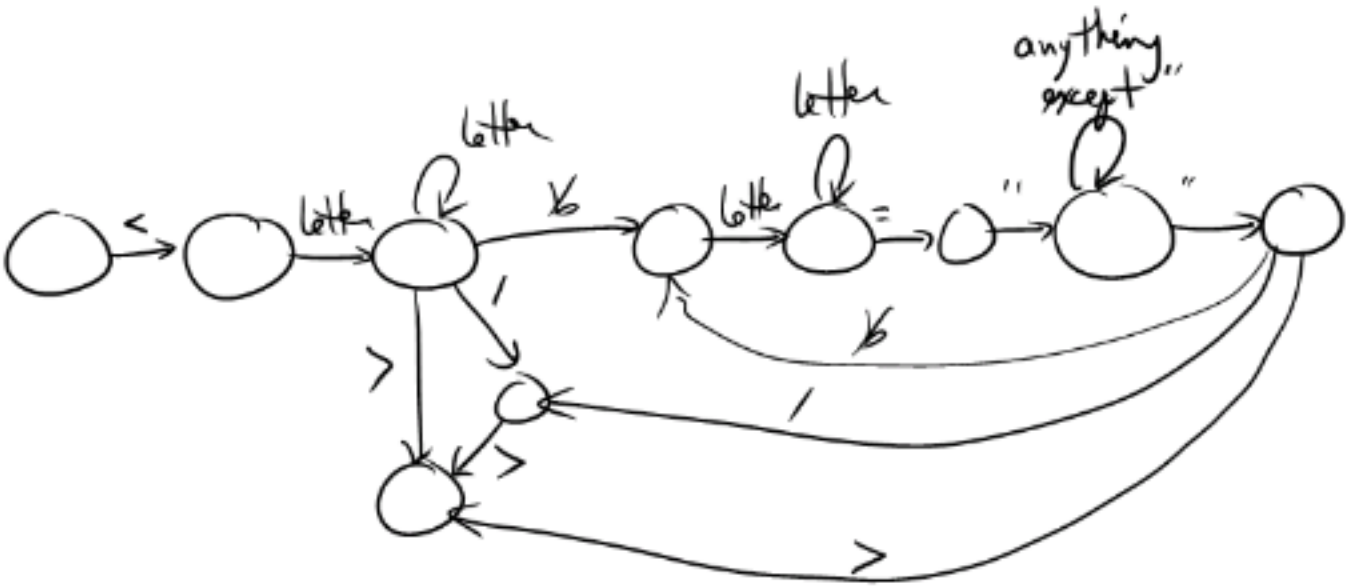
```
<script language="javascript" version="1.0">
```

Write a regular expression and a DFA for XML nodes.

word = ['a' - 'z']+

atlr = ' ' word "=\"" [^\"]* "\""

'<' word atlr* '/'? '>'

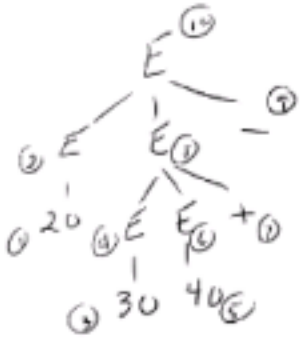


4. (*Bottom-up parsing, including the mechanics of shift/reduce parsing. Not covered here are the issues that arise with ambiguity, precedence, and associativity, and how they are handled by ocamlyacc declarations; you should study those as well.*)

Consider this grammar for "postfix notation":

```
E -> int  |  E E +  | E E -
```

This grammar is not ambiguous. Here is a parse tree for the sentence "20 30 40 + -", with the nodes labelled in pre-order:



Write out the shift-reduce parse for this sentence:

	Action	Stack	Input
1.	Sh		20 30 40 + -
2.	R E→int	20	30 40 + -
3.	Sh	E	30 40 + -
4.	R E→int	E 30	40 + -
5.	Sh	E E	40 + -
6.	R E→int	E E 40	+ -
7.	Sh	E E E	+ -
8.	R E→E E +	E E E +	-
9.	Sh	E E	-
10.	R E→E E -	E E -	eof
11.	Accept	E	eof

5. (*This question is from midterm 1.*)

Consider these two expression grammars:

G1

$A \rightarrow B \mid B * A \mid B / A$
$B \rightarrow C \mid B + C \mid B - C$
$C \rightarrow \text{id} \mid \text{int}$

G2

$A \rightarrow \text{id} \mid \text{int} \mid A{+}A \mid A{-}A \mid A{*}A \mid A/A$

(a) (8 pts) Draw the parse tree for `x+y*10` in G1:

(b) (6 pts) What *precedences* and *associativities* are enforced by G1, if any?

+, − prec. over *, /

+, − left-assoc

*, / right-assoc

(c) (6 pts) Provide ocamlyacc precedence declarations for G2 so that the precedence and associativity of all operators is the same as those enforced by G1. (For tokens, use: `Star`, `Slash`, `Plus`, `Minus`, `Id of string`, and `Int of int`.)

%right Star Slash

%left Plus ~~Star~~ Minus

6. (*Top-down, or recursive-descent, parsing. This includes calculating FIRST and FOLLOW sets, determining whether a grammar is LL(1), and writing a top-down parser.*)

Consider this grammar:

```
expr  ->  funcall  |  varref
varref  ->  ident  |   ident . varref
funcall  ->  ident ( expr)  |  ident . funcall
```

(a) Give one reason why this grammar is not LL(1).

Two rhs's for varref start the same
(Have two prods varref → α, varref → β, FIRST(α) ∩ FIRST(β) ≠ ∅)

(b) This grammar results from the one above by a common transformation:

```
expr  ->  funcall  |  varref
varref  ->  ident dotvarref
dotvarref ->  epsilon  |  . varref
funcall  ->  ident dotfuncall
dotfuncall ->  ( expr )  |  . funcall
```

What is the name of the transformation we used? Left-factoring

Give one reason why this grammar is still not LL(1)

For prod's expr → funcall, expr → varref,
FIRST(funcall) ∩ FIRST(varref) ≠ ∅

(c) This is yet another variant on the above grammar:

```
expr  ->  varref funcallopt
varref  ->  ident dotvarref
dotvarref ->  epsilon  |  . varref
funcallopt  -> ( expr )  |  epsilon
```

expr → varref funcallopt $

Calculate the FIRST and FOLLOW sets for this grammar:

	FIRST 0	1	2	3	FOLLOW 0	1	2	3	4
e	∅	∅	id	id	eof	eof,)	eof,)	eof,)	eof,)
v	∅	id	id	id	∅	(, eof	(, eof,)	(, eof,)	(, eof,)
d	∅	•, .	•, .	•, .	∅	∅	(, eof	(, eof,)	(, eof,)
f	∅	•, (	•, (	•, (	∅	eof	eof,)	eof)	eof,)

(d) Is the grammar LL(1)? Explain. FIRST(. varref) ∩ FOLLOW(dotvarref) ?= ∅

FIRST ((expr)) ∩ FOLLOW(funcallopt) = ∅

{ (} {eof,) }

7. (*Understanding evaluation (SOS) rules, including writing out a full evaluation tree for an expression.*)

Here are the evaluation rules for a subset of OCaml (the same as μOCaml from midterm 2), in three models. (Note that the lazy evaluation rules are identical to the substitution rules except for App.)

Substitution model

(Const) Int x $\Downarrow$ Int x

(Fun) Fun(a,e) $\Downarrow$ Fun(a,e)

(δ) e *op* e' $\Downarrow$ v *OP* v'
$e \Downarrow v$
$e' \Downarrow v'$

(App) $e\ e' \Downarrow v$
$e \Downarrow$ Fun(a, e'')
$e' \Downarrow v'$
$e''[v'/a] \Downarrow v$

Environment model

(Const) Int i, $\rho \Downarrow$ Int i

(Var) a, $\rho \Downarrow \rho(a)$

(δ) e *op* e', $\rho \Downarrow v$ *OP* v'
e, $\rho \Downarrow v$
e', $\rho \Downarrow v'$

(App) $e\ e'$, $\rho \Downarrow v$
e, $\rho \Downarrow <$Fun(a, e''), $\rho'>$
e', $\rho \Downarrow v'$
e'', $\rho'[a \mapsto v'] \Downarrow v$

(Fun) Fun(a,e), $\rho \Downarrow <$Fun(a,e), $\rho>$

Substitution model, with lazy evaluation

(Const) Int x $\Downarrow_\ell$ Int x

(Fun) Fun(a,e) $\Downarrow_\ell$ Fun(a,e)

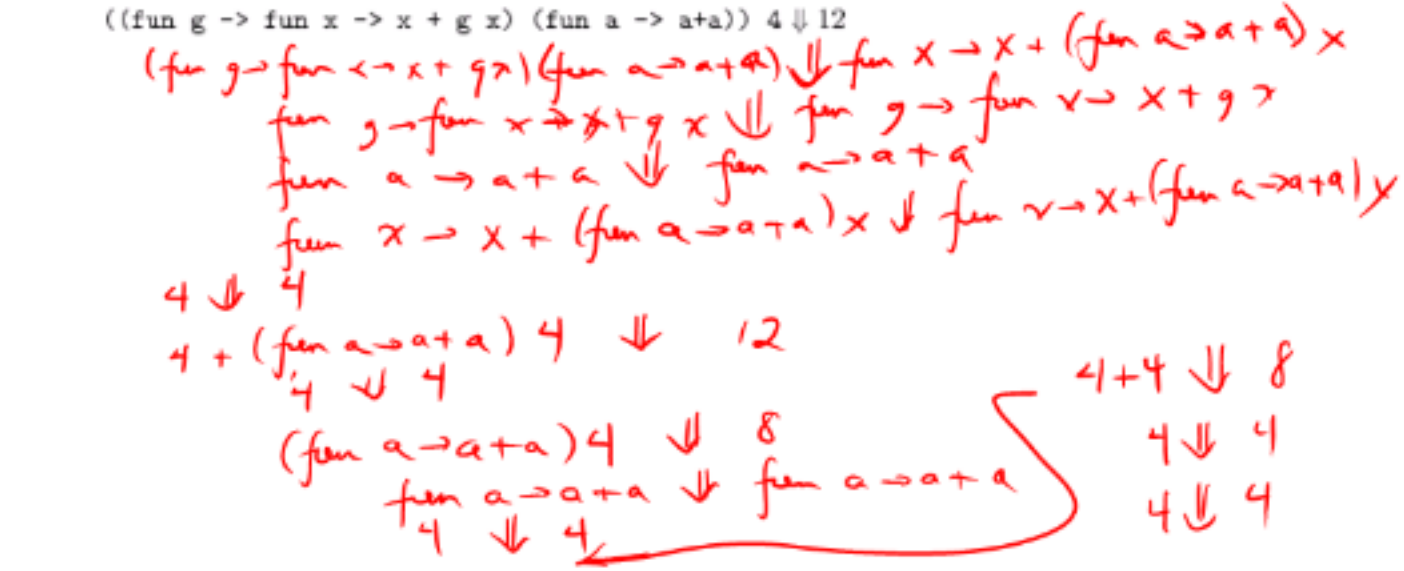
(δ) e *op* e' $\Downarrow_\ell$ v *OP* v'
$e \Downarrow_\ell v$
$e' \Downarrow_\ell v'$

(App) $e\ e' \Downarrow_\ell v$
$e \Downarrow_\ell$ Fun(a, e'')
$e''[e'/a] \Downarrow_\ell v$

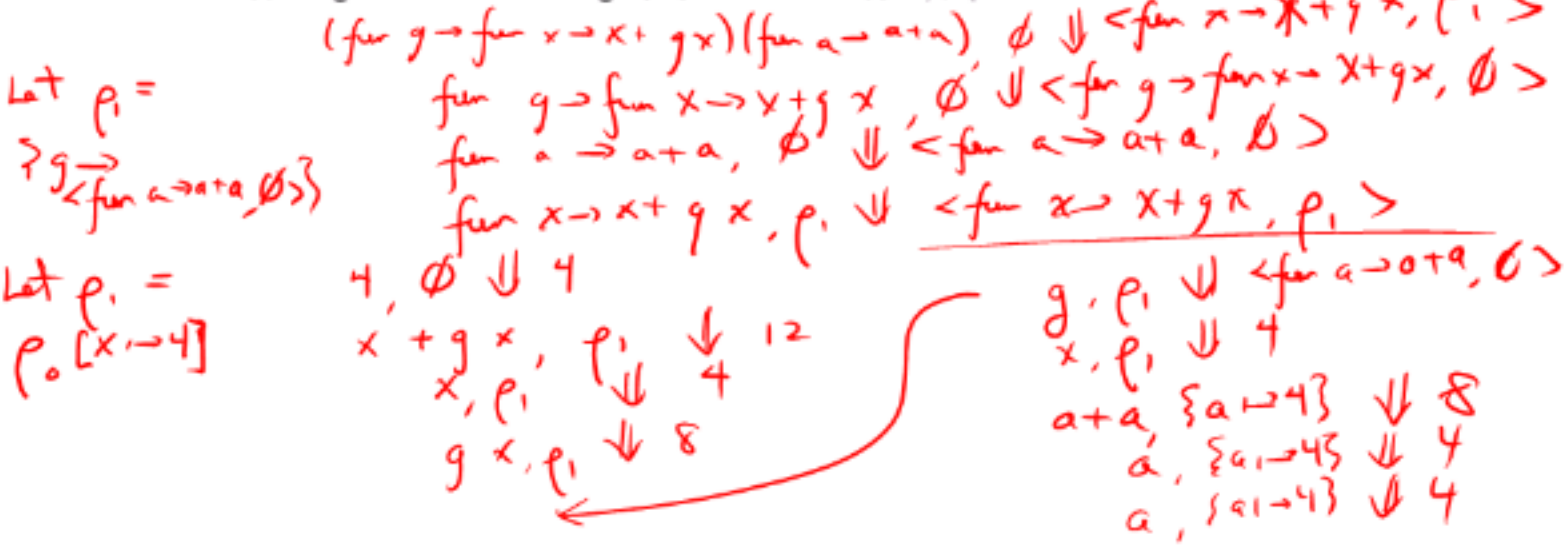
(a) On the next page, write the evaluation tree for the following expression in each of the three models:

```
((fun g -> fun x -> x + g x) (fun a -> a+a)) 4
```

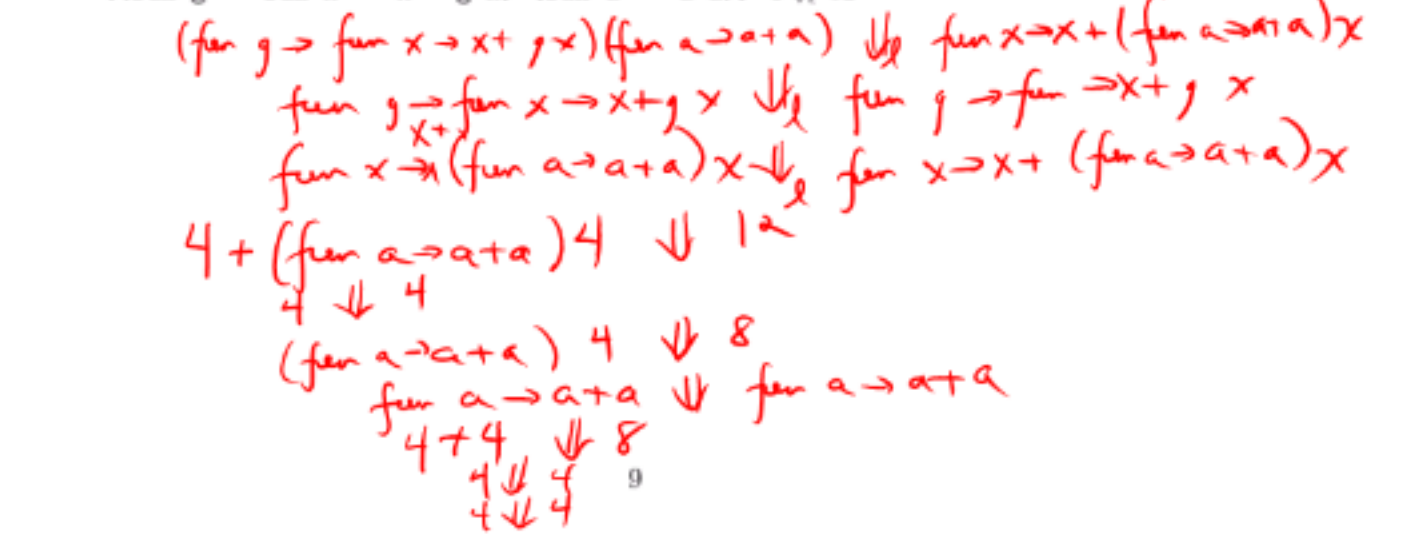
((fun g -> fun x -> x + g x) (fun a -> a+a)) 4 ⇓ 12



((fun g -> fun x -> x + g x) (fun a -> a+a)) 4, ∅ ⇓ 12



((fun g -> fun x -> x + g x) (fun a -> a+a)) 4 ⇓$_\ell$ 12



8. (*Understanding evaluation (SOS) rules to the extent of being able to write them yourself.*)

To add lists, we add three new expressions: `[]`, e `::` e', and a simple version of `match`: `match` e `with []` `->` e' `|` x`::`y `=` e''. In this version of `match`, e should evaluate to a value of the form `[]` or v`::`v', where v and v' are both values. We give the rule for the first two expressions in each model. There are two rules for `match` in each model, one for when e evaluates to an empty list and one for when it evaluates to a non-empty list. Give both rules in both models:

(Null) $[] \Downarrow []$

(Cons) $\dfrac{e \Downarrow v \quad e' \Downarrow v'}{e :: e' \Downarrow v :: v'}$

(MatchNull) `match` e `with []` `->` e' `|` x`::`y `->` $e'' \Downarrow v$

$$\frac{e \Downarrow [] \quad e' \Downarrow v}{\texttt{match } e \texttt{ with } [] \texttt{ -> } e' \texttt{ | } x\texttt{::}y \texttt{ -> } e'' \Downarrow v}$$

(MatchCons) `match` e `with []` `->` e' `|` x`::`y `->` $e'' \Downarrow v$

$$\frac{e \Downarrow v_1 :: v_2 \quad e''[v_1/x][v_2/y] \Downarrow v}{\texttt{match } e \texttt{ with } [] \texttt{ -> } e' \texttt{ | } x\texttt{::}y \texttt{ -> } e'' \Downarrow v}$$

(Null) $[], \rho \Downarrow []$

(Cons) $\dfrac{e, \rho \Downarrow v \quad e', \rho \Downarrow v'}{e :: e', \rho \Downarrow v :: v'}$

(MatchNull) `match` e `with []` `->` e' `|` x`::`y `->` $e'', \rho \Downarrow v$

$$\frac{e, \rho \Downarrow [] \quad e', \rho \Downarrow v}{\texttt{match } e \texttt{ with } [] \texttt{ -> } e' \texttt{ | } x\texttt{::}y \texttt{ -> } e'', \rho \Downarrow v}$$

(MatchCons) `match` e `with []` `->` e' `|` x`::`y `->` $e'', \rho \Downarrow v$

$$\frac{e, \rho \Downarrow v_1 :: v_2 \quad e'', \rho[x \mapsto v_1, y \mapsto v_2] \Downarrow v}{\texttt{match } e \texttt{ with } [] \texttt{ -> } e' \texttt{ | } x\texttt{::}y \texttt{ -> } e'', \rho \Downarrow v}$$