CS421 Summer 2008 Midterm Solutions

CS 421 — Programming Languages and Compilers
Summer Term 2008
11:30am–12:45pm, Monday, July 7, 2008

The total time for this exam is 75 minutes.

Print your name and netid below. Also write your name at the bottom of each subsequent page.

Name:

Netid:

- This is a **closed-book** exam. You are allowed one 3 inch by 5 inch card of notes prepared by yourself. You may write on both sides of the card. This card is **not to be shared**. All other materials, besides pens, pencils, and erasers, are to be put away.
- Do not share anything with other students. Do not talk to other students. Do not look at another student's exam. Also, be careful to 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, you must document your assumptions about what the question means. We are not allowed to answer questions about the exam in class, and obviously cannot answer these questions for students taking the exam elsewhere.
- Including this cover sheet, rules at the end, and scratch pages, there are 13 pages to the exam. Please verify that you have all 13 pages. You are allowed to tear off the last three (rules and scratch) if this makes it easier to take the test.

Question	Total points	Score
1	10	
2	15	
3	15	
4	8	
5	12	
6	20	
7	20	
SUBTOTAL	100	
Extra Credit	12	
TOTAL	112	

1. **Short Answer Questions** (*10 points*):

   (a) *In the following, state TRUE or FALSE for each part separately; each part is 1 point* (*write your answers to the left*):

   A typical dynamically-typed language:

   i. assigns types to all expressions at compile time **FALSE**
   ii. allows variables to hold values of different types at runtime **TRUE**
   iii. always detects all potential type errors in a program **FALSE**
   iv. checks that correct types are used in an expression when that expression is executed **TRUE**

   (b) *In the following, state TRUE or FALSE for each part separately; each part is 1 point* (*write your answers to the left*):

   Lexing and parsing:

   i. any language construct that can be defined with a regular expression can also be defined with a context-free grammar **TRUE**
   ii. any language construct that can be defined with a context-free grammar can also be defined with a regular expression **FALSE**
   iii. an LL(1) parser can be created for any context-free grammar **FALSE**
   iv. LR parsing techniques are strictly more powerful than LL parsing techniques **TRUE**

   (c) (*2 points*) What value is produced by the following program?

```
let x = 5 and y = 10
  in let y = x + y
    in let f x = x + y
      in let x = y + x
        in f x
```

   35

2. **Functional Programming** (*15 points*):

(a) (*3 points*) Write an OCaml function `pairs` : $'$`a` $\rightarrow$ $'$`b list` $\rightarrow$ ($'$`a` $*'$ `b`) `list` that, given an item `a` and a list of items `b`, returns a new list, where each `b` originally in the list is replaced by the pair `(a,b)` – e.g., where `pairs a [i;j]` becomes `[ (a,i) ; (a,j) ]`.

```
let rec pairs a bs =
  match bs with
    | [] -> []
    | x::xs -> (a,x) :: pairs a xs
```

(b) (*3 points*) What is the result of `pairs` 3 $[5; 10; 15]$?

```
# pairs 3 [5;10;15];;
- : (int * int) list = [(3, 5); (3, 10); (3, 15)]
```

(c) (*3 points*) What is the result of `pairs` 3 [“5”; “10”; “15”]?

```
# pairs 3 ["5";"10";"15"];;
- : (int * string) list = [(3, "5"); (3, "10"); (3, "15")]
```

(d) (*3 points*) What is the result (English-language description) of `pairs` 3?

```
# pairs 3 ;;
- : '_a list -> (int * '_a) list = <fun>
```

A function that, given a list of items, will return a new list made up of those items paired with 3 (pairs of the form $(3, \texttt{b})$, where b is an item from the list).

(e) (*3 points*) What is the result of evaluating
`List.fold_left` (`fun x y` $->$ `let`$(\texttt{a}, \texttt{b}) = \texttt{y in x} + \texttt{a} + \texttt{b}$) 0 (`pairs` 1 $[2; 3; 4]$)?

```
# List.fold_left (fun x y -> let (a,b) = y in x + a + b) 0 (pairs 1 [2;3;4]) ;;
- : int = 12
```

Note: This is the function that sums up all the pairs.

3. **Higher-Order Functions and Data Types** (*15 points*):

You are allowed to use all functions in `List`, such as `List.rev`, `List.fold_left`, etc – you do not have to write them yourself.

Given the following OCaml datatype:

```
type list_tree = Empty | Node of list_tree * list_tree | Leaf of int list;;
```

(a) (*5 points*) Write a recursive function `fold_list_tree`:

```
val fold_list_tree : ('a -> 'a -> 'a) -> (int list -> 'a) -> 'a -> list_tree -> 'a = <fun>
```

that folds a function over a `list_tree`. It should take a function that works over `Node`s (typed between the first set of parens in the type signature for `fold_list_tree`), a function that works over `Leaf`s (typed between the second set of parens in the type signature for `fold_list_tree`), and an identity for `Empty`, as well as the `list_tree`, returning the value of the fold computation.

```
# let rec fold_list_tree nf lf i tree =
    match tree with
      | Empty -> i
      | Node (lt,rt) -> nf (fold_list_tree nf lf i lt) (fold_list_tree nf lf i rt)
      | Leaf il -> lf il
val fold_list_tree :
  ('a -> 'a -> 'a) -> (int list -> 'a) -> 'a -> list_tree -> 'a = <fun>
```

(b) (*5 points*) Write a function `prod_list_tree`, using `fold_list_tree`, that will calculate the product of all the integers stored in the integer lists in the `Leaf` nodes of the `list_tree`.

```
# let prod_list_tree lt =
    fold_list_tree (fun x y -> x * y) (fun il -> List.fold_left ( * ) 1 il) 1 lt
val prod_list_tree : list_tree -> int = <fun>
# lt1;;
- : list_tree =
Node (Node (Leaf [1; 2; 3], Empty),
 Node (Node (Leaf [4; 5; 6], Leaf [7; 8; 9]), Node (Empty, Leaf [])))
# prod_list_tree lt1;;
- : int = 362880
```

(c) (*5 points*) Write a function `flip_list_tree`, using `fold_list_tree`, that will switch the left and right trees in any `Node` and reverse the lists stored in any `Leaf`.

```
# let flip_list_tree lt =
    fold_list_tree (fun x y -> Node (y, x)) (fun il -> Leaf (List.rev il)) Empty lt;;
val flip_list_tree : list_tree -> list_tree = <fun>
# lt1;;
- : list_tree =
Node (Node (Leaf [1; 2; 3], Empty),
 Node (Node (Leaf [4; 5; 6], Leaf [7; 8; 9]), Node (Empty, Leaf [])))
# flip_list_tree lt1;;
- : list_tree =
Node (Node (Node (Leaf [], Empty), Node (Leaf [9; 8; 7], Leaf [6; 5; 4])),
 Node (Empty, Leaf [3; 2; 1]))
```

4. **Regular Expressions** (*8 points*):

(a) (*4 points*) Over the alphabet $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, give a regular expression generating exactly those strings over $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ such that no `b` occurs after the final `c`. Remember, this does not mean the string needs to contain a `c`!

$$(\mathtt{a} + \mathtt{b})^* + ((\mathtt{a} + \mathtt{b} + \mathtt{c})^* \mathtt{c} \mathtt{a}^*)$$

(b) (*4 points*) Over the alphabet $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, give a regular expression generating exactly those strings over $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ where any `a` in the string occurs before any `b` but after at least one `c` (i.e. if there is an `a` in the string, there is at least one `c` before it; a mix of `a` and `c` characters can then occur, followed by a mix of `b` and `c` characters; if there is no `a`, `c` is allowed but not required.).

$$(\mathtt{c}(\mathtt{c} + \mathtt{a})^*(\mathtt{c} + \mathtt{b})^*) + (\mathtt{b} + \mathtt{c})^*$$

5. **Variables and Scope** (*12 points*): You are given the following program in a standard block-structured language:

```
program main;
var x : integer;
var y : integer;

procedure f()
var y : integer;
var z : integer;
begin (* code in procedure f *)
  x := 20;  y := 30;  z := 50;
  g();
end (* code in procedure f *)

procedure g()
var x : integer;
begin (* code in procedure g *)
  x := 40;
  print("The value of x is");
  printint(x);
  print("The value of y is"); (* POINT *)
  printint(y);
  x := y + z; (* POINT 2 *)
end (* code in procedure g *)

begin (* main body of program *)
  x := 10; y := 15;
  f();
end (* main body of program *)
```

(a) (*3 points*) Assuming static scope, what is the referencing environment at the point marked `POINT` (reminder: this is the set of all variables visible at the marked point, with a prefix indicating where it was defined, like `main.x` or `f.z`)? Assume `POINT` is reached by the main program body invoking `f`, which then invokes `g`, in this and the next two problems.

$\{\texttt{g.x}, \texttt{main.y}\}$

(b) (*3 points*) Assuming static scope, what will be printed by the `printint(y)` statement (which just outputs the current value of `y`) on the line after `POINT`?

15 will be printed, since it is `main.y` that is printed here.

(c) (*3 points*) Assuming dynamic scope, what will be printed by the `printint(y)` statement on the line after `POINT`?

30 will be printed, since it is `f.y` that is printed here.

(d) (*3 points*) Assume `POINT 2` is reached by the main program body invoking `f`, which then invokes `g`. Is this line of the program valid under static scoping? Under dynamic?

This line is **NOT** valid under static scoping, since `z` will not be in scope. It **IS** valid under dynamic scoping; `z` will be in scope since it was declared in `f` and was still in scope in `f` when `g` was invoked.

6. **Context-Free Grammars, Derivations, and Parse Trees** (*20 points*): The following exercises make use of this grammar for arithmetic expressions. Here, the start symbol is E, `id` refers to identifiers made up of one lowercase character ($\texttt{a}, \texttt{b}, \cdots, \texttt{y}, \texttt{z}$), and `num` refers to any integer:

$$\begin{array}{lll} E \to E + T & T \to T * F & F \to \texttt{id} \\ E \to E - T & T \to T/F & F \to \texttt{num} \\ E \to T & T \to F & F \to (E) \end{array}$$

(a) (*5 points*) Show a leftmost derivation for the following term:

$$x * y + (5 - 3)$$

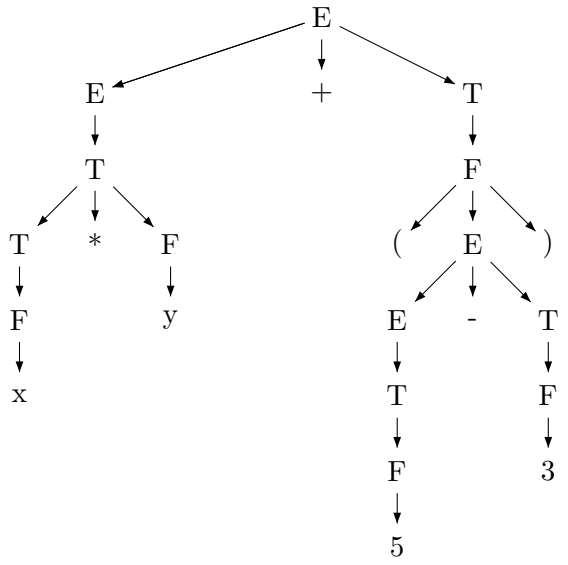
$$E \Rightarrow E + T \Rightarrow T + T \Rightarrow T * F + T \Rightarrow F * F + T \Rightarrow x * F + T \Rightarrow x * y + T \Rightarrow$$
$$x * y + F \Rightarrow x * y + (E) \Rightarrow x * y + (E - T) \Rightarrow x * y + (T - T) \Rightarrow x * y + (F - T) \Rightarrow$$
$$x * y + (5 - T) \Rightarrow x * y + (5 - F) \Rightarrow x * y + (5 - 3)$$

(b) (*5 points*) Show a rightmost derivation for the same term:

$$E \Rightarrow E + T \Rightarrow E + F \Rightarrow E + (E) \Rightarrow E + (E - T) \Rightarrow E + (E - F) \Rightarrow E + (E - 3) \Rightarrow$$
$$E + (T - 3) \Rightarrow E + (F - 3) \Rightarrow E + (5 - 3) \Rightarrow T + (5 - 3) \Rightarrow T * F + (5 - 3) \Rightarrow$$
$$T * y + (5 - 3) \Rightarrow F * y + (5 - 3) \Rightarrow x * y + (5 - 3)$$

$E \to E + T$	$T \to T * F$	$F \to \texttt{id}$
$E \to E - T$	$T \to T/F$	$F \to \texttt{num}$
$E \to T$	$T \to F$	$F \to (E)$

(c) (*6 points*) Provide a parse tree for the same term, $x * y + (5 - 3)$, using the canonical derivation. *Note here that we don't need to bother with the derivation order here, since the grammar is unambiguous – if this grammar were ambiguous, the derivation would matter, since we could get different parse trees*:



(d) (*2 points, circle the answer*) Is there a unique parse tree for this term? **True**/**False** If this were not true, then we would have two different ways of interpreting this term in the language.

(e) (*2 points, circle the answer*) Is there a unique leftmost derivation for this term? **True**/**False** If this were not true, we would have an ambiguous grammar.

7. **Type Derivations** (*20 points*):
Below is an outline of the type derivation of the following expression:

```
if b then ((fun x -> x) 4) else (g 5)
```

Please complete the derivation by giving the results to go in the numbered blanks. Put your answers next to the corresponding numbers below the type derivation outline. Feel free to use $\Gamma = \{\texttt{g} : \texttt{int} \to \texttt{int},\ \texttt{b} : \texttt{bool}\}$ when listing the type environment. You can find the type derivation rules at the back of your exam; feel free to tear that page out for easier reference.

$$\dfrac{\dfrac{}{\Gamma \vdash \texttt{b} : \texttt{bool}} \quad \dfrac{\dfrac{\dfrac{}{\Gamma \cup (\texttt{x} : \texttt{int}) \vdash \texttt{x} : \texttt{int}}}{\Gamma \vdash \texttt{fun x -> x} : \texttt{int} \to \texttt{int}} \quad \dfrac{}{\Gamma \vdash \texttt{4} : \texttt{int}}}{\Gamma \vdash \texttt{(fun x -> x) 4} : \texttt{int}} \quad \dfrac{\dfrac{}{\Gamma \vdash \texttt{g} : \texttt{int} \to \texttt{int}} \quad \dfrac{}{\Gamma \vdash \texttt{5} : \texttt{int}}}{\Gamma \vdash \texttt{g 5} : \texttt{int}}}{\{\texttt{g} : \texttt{int} \to \texttt{int},\ \texttt{b} : \texttt{bool}\} \vdash \texttt{if b then ((fun x -> x) 4) else (g 5)} : \texttt{int}}$$

#1 `bool`

#2 Γ

#3 `(fun x -> x) 4`

#4 `int`

#5 Γ

#6 `int`

#7 Γ

#8 `fun x -> x`

#9 `int` $\to$ `int`

#10 Γ

#11 `4`

#12 Γ

#13 `g`

#14 `int` $\to$ `int`

#15 Γ

#16 `5`

#17 `int`

#18 $\Gamma \cup$ `(x : int)`

#19 `x`

#20 `int`

8. **Extra Credit: Unification** (*12 points*):

(a) (*10 points*) Give a most general unifier for the following unification problem. Lower case letters (f, g, h, a, b, c) are constants or term constructors: specifically, f is a term constructors with arity 2, g and h are term constructors with arity 1, and a, b, and c are constant terms with arity 0. Letters α,β, δ, and γ are variables. Show your work by listing the operation performed in each step of unification – decompose, orient, delete, eliminate – and the result of the step. If unification is not possible, work as far as possible and show where unification fails. If unification does not fail, show the final substitution, which should be a set of variable to term mappings.

$$\{\mathtt{f}(\mathtt{f}(\alpha,\mathtt{g}(\mathtt{b})),\mathtt{g}(\beta)) = \mathtt{f}(\gamma,\mathtt{g}(\mathtt{a}))\ ;\ \mathtt{g}(\alpha) = \mathtt{g}(\mathtt{h}(\delta))\ ;\ \mathtt{f}(\mathtt{h}(\delta),\gamma) = \mathtt{f}(\mathtt{h}(\mathtt{c}),\gamma)\}$$

decompose	$\{\mathtt{f}(\alpha,\mathtt{g}(\mathtt{b})) = \gamma\ ;\ \mathtt{g}(\beta) = \mathtt{g}(\mathtt{a})\ ;\ \mathtt{g}(\alpha) = \mathtt{g}(\mathtt{h}(\delta))\ ;\ \mathtt{f}(\mathtt{h}(\delta),\gamma) = \mathtt{f}(\mathtt{h}(\mathtt{c}),\gamma)\}$
orient	$\{\gamma = \mathtt{f}(\alpha,\mathtt{g}(\mathtt{b}))\ ;\ \mathtt{g}(\beta) = \mathtt{g}(\mathtt{a})\ ;\ \mathtt{g}(\alpha) = \mathtt{g}(\mathtt{h}(\delta))\ ;\ \mathtt{f}(\mathtt{h}(\delta),\gamma) = \mathtt{f}(\mathtt{h}(\mathtt{c}),\gamma)\}$
decompose	$\{\gamma = \mathtt{f}(\alpha,\mathtt{g}(\mathtt{b}))\ ;\ \beta = \mathtt{a}\ ;\ \mathtt{g}(\alpha) = \mathtt{g}(\mathtt{h}(\delta))\ ;\ \mathtt{f}(\mathtt{h}(\delta),\gamma) = \mathtt{f}(\mathtt{h}(\mathtt{c}),\gamma)\}$
decompose	$\{\gamma = \mathtt{f}(\alpha,\mathtt{g}(\mathtt{b}))\ ;\ \beta = \mathtt{a}\ ;\ \alpha = \mathtt{h}(\delta)\ ;\ \mathtt{f}(\mathtt{h}(\delta),\gamma) = \mathtt{f}(\mathtt{h}(\mathtt{c}),\gamma)\}$
decompose	$\{\gamma = \mathtt{f}(\alpha,\mathtt{g}(\mathtt{b}))\ ;\ \beta = \mathtt{a}\ ;\ \alpha = \mathtt{h}(\delta)\ ;\ \mathtt{h}(\delta) = \mathtt{h}(\mathtt{c})\ ;\ \gamma = \gamma\ \}$
delete	$\{\gamma = \mathtt{f}(\alpha,\mathtt{g}(\mathtt{b}))\ ;\ \beta = \mathtt{a}\ ;\ \alpha = \mathtt{h}(\delta)\ ;\ \mathtt{h}(\delta) = \mathtt{h}(\mathtt{c})\ \}$
decompose	$\{\gamma = \mathtt{f}(\alpha,\mathtt{g}(\mathtt{b}))\ ;\ \beta = \mathtt{a}\ ;\ \alpha = \mathtt{h}(\delta)\ ;\ \delta = \mathtt{c}\ \}$
eliminate	$\{\gamma = \mathtt{f}(\alpha,\mathtt{g}(\mathtt{b}))\ ;\ \beta = \mathtt{a}\ ;\ \alpha = \mathtt{h}(\mathtt{c})\ ;\ \delta = \mathtt{c}\ \}$
eliminate	$\{\gamma = \mathtt{f}(\mathtt{h}(\mathtt{c}),\mathtt{g}(\mathtt{b}))\ ;\ \beta = \mathtt{a}\ ;\ \alpha = \mathtt{h}(\mathtt{c})\ ;\ \delta = \mathtt{c}\ \}$
FINAL	$\{\gamma = \mathtt{f}(\mathtt{h}(\mathtt{c}),\mathtt{g}(\mathtt{b}))\ ;\ \beta = \mathtt{a}\ ;\ \alpha = \mathtt{h}(\mathtt{c})\ ;\ \delta = \mathtt{c}\ \}$

(b) (*2 points*) Using the unifier discovered above, apply the substitution and show the final terms, which should be equalities and which should have no variables. If unification got stuck above, just write "unification failed" below.

- $\mathtt{f}(\mathtt{f}(\alpha,\mathtt{g}(\mathtt{b})),\mathtt{g}(\beta)) = \mathtt{f}(\gamma,\mathtt{g}(\mathtt{a})) \rightarrow \mathtt{f}(\mathtt{f}(\mathtt{h}(\mathtt{c}),\mathtt{g}(\mathtt{b})),\mathtt{g}(\mathtt{a})) = \mathtt{f}(\mathtt{f}(\mathtt{h}(\mathtt{c}),\mathtt{g}(\mathtt{b})),\mathtt{g}(\mathtt{a}))$
- $\mathtt{g}(\alpha) = \mathtt{g}(\mathtt{h}(\delta)) \rightarrow \mathtt{g}(\mathtt{h}(\mathtt{c})) = \mathtt{g}(\mathtt{h}(\mathtt{c}))$
- $\mathtt{f}(\mathtt{h}(\delta),\gamma) = \mathtt{f}(\mathtt{h}(\mathtt{c}),\gamma) \rightarrow \mathtt{f}(\mathtt{h}(\mathtt{c}),\mathtt{f}(\mathtt{h}(\mathtt{c}),\mathtt{g}(\mathtt{b}))) = \mathtt{f}(\mathtt{h}(\mathtt{c}),\mathtt{f}(\mathtt{h}(\mathtt{c}),\mathtt{g}(\mathtt{b})))$

Rules for type derivations:

Constants

$$\frac{}{\vdash n : \texttt{int}} \text{(assuming } n \text{ is an int)}$$

$$\frac{}{\vdash \texttt{true} : \texttt{bool}}$$

$$\frac{}{\vdash \texttt{false} : \texttt{bool}}$$

Variables

$$\frac{}{\Gamma \vdash x : \tau} \text{ if } (x : \tau) \in \Gamma$$

Arithmetic Operators

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \oplus e_2 : \texttt{int}} (\oplus \in \{+, -, *, /, \cdots\})$$

Relational Operators

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \sim e_2 : \texttt{bool}} (\sim \in \{<, >, \leq, \geq, =, \neq, \cdots\})$$

Booleans

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \quad \Gamma \vdash e_2 : \texttt{bool}}{\Gamma \vdash e_1 \texttt{ \&\& } e_2 : \texttt{bool}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \quad \Gamma \vdash e_2 : \texttt{bool}}{\Gamma \vdash e_1 \texttt{ || } e_2 : \texttt{bool}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool}}{\Gamma \vdash \texttt{! } e_1 : \texttt{bool}}$$

If

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : \tau}$$

Function Abstraction

$$\frac{\Gamma \cup [\texttt{x} : \tau_1] \vdash e : \tau_2}{\Gamma \vdash \texttt{fun x->} e : \tau_1 \to \tau_2}$$

Function Application

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 \ e_2 : \tau_2}$$

Let

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \cup [x : \tau] \vdash e_2 : \tau'}{\Gamma \vdash \texttt{let x} = e_1 \texttt{ in } \ e_2 : \tau'}$$

Letrec

$$\frac{\Gamma \cup [x : \tau] \vdash e_1 : \tau \quad \Gamma \cup [x : \tau] \vdash e_2 : \tau'}{\Gamma \vdash \texttt{let rec x} = e_1 \texttt{ in } \ e_2 : \tau'}$$

Scratch Page 1

Scratch Page 2