

# CS421 Spring 2008 Midterm

Wednesday, February 27, 2008

Name:	
NetID:	

- You have **60 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, you may seek clarification from one of the TAs. You must use a whisper, or write your question out. Speaking out aloud is not allowed.
- Including this cover sheet, there are 12 pages to the exam. Please verify that you have all 12 pages.
- Please write your name and NetID in the spaces above, and also at the top of every page.

Question	Possible points	Points earned
1	6	
2	12	
3	6	
4	8	
5	8	
6	10	
7	10	
8	10	
9	10	
10	10	
11	10	
Total	100	

1. (6 pts) Give the types of each of the following Ocaml functions:

(a) `let alwaysfour x = 4`

**val alwaysfour : 'a -> int**

(b) `let add x y = x + y`

**val add : int -> int -> int**

(c) `let concat x y = x ^ y`

**val concat : string -> string -> string**

(d) `let addmult x y = (x + y, x * y)`

**val addmult : int -> int -> int \* int**

(e) `let rec f x = if x=[] then [] else hd x @ f (tl x)`

**val f : 'a list list -> 'a list**

(f) `let rec copy x = if x=[] then [] else hd x :: copy (tl x)`

**val copy : 'a list -> 'a list**

2. (12 pts) Define the following OCaml functions:

(a) `contains : 'a -> 'a list -> bool` such that `contains x lst` returns true if and only if `lst` has `x` as one of its elements. Do not use any pre-existing functions. E.g.

```
contains 4 [3;4;5] = true
```

```
let rec contains x lst =
  match lst with
  [] -> false
  | y::ys -> x = y || contains x ys
```

(b) `evens : 'a list -> 'a list` returns the 2nd, 4th, etc. elements of its argument. E.g.

```
evens [13;5;9;0;7;8] = [5; 0; 8]
```

```
let rec evens lis = match lis with
  [] -> []
  | [a] -> []
  | (a::b::lis') -> b :: evens lis'
```

(c) Implement the OCaml function `partition : int list -> (int list) list`, which divides a list into “runs” of the same integer, e.g.

```
partition [9;9;5;6;6;6;3] = [[9;9]; [5]; [6;6;6]; [3]]
```

(You may define auxiliary functions, but it is not actually necessary.)

```
let rec partition lis =
  if lis = [] then []
  else match partition (tl lis) with
  [] -> [[hd lis]]
  | x :: xs -> if hd lis = hd x
  then (hd lis :: x) :: xs
  else [hd lis] :: (x :: xs)
```

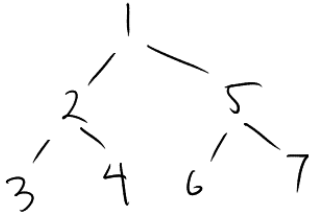
3. (6 pts) Given type definition

```
type `a tree = Node of `a * (`a tree) list
```

we define a tree as a labeled node with zero or more children. For example,

```
let t1 = Node(1, [Node(2, [Node(3, []); Node(4, [])]);
                 Node(5, [Node(6, []); Node(7, [])])])
```

represents the tree



Define the mutually recursive functions `preorder : `a tree -> `a list` and `preorder_list : (`a tree) list -> `a list` such that `preorder t` gives the pre-order traversal of `t`, and `preorder_list` appends the pre-order traversals of its component trees together. Your functions should be mutually recursive. You may use `@` or `append`.

E.g. `preorder t1 = [1;2;3;4;5;6;7]`

```
let rec preorder (Node(x, ts)) =
```

```
  x::preorder_list ts
```

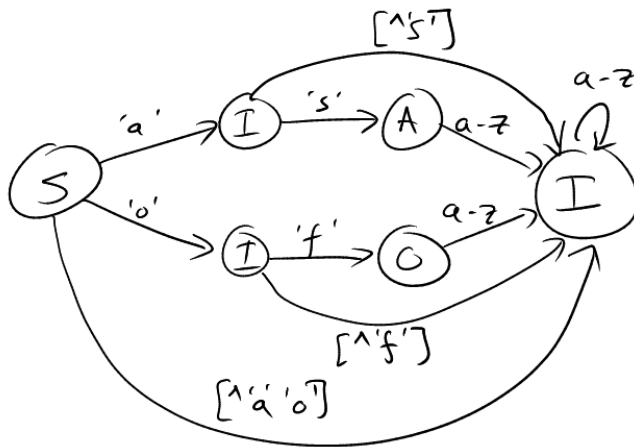
```
and preorder_list lst =
```

```
  match lst with
```

```
    [] -> []
```

```
  | t::ts -> preorder t @ preorder_list ts
```

4. (8 pts) An identifier is any sequence of one or more characters 'a' - 'z' that is not a keyword. Our language recognizes identifiers and the keywords 'of' and 'as'. Give a deterministic finite-state machine that recognizes our language. Each state should be labeled with either S (start state), O ('of' keyword), A ('as' keyword) or I (identifier).



5. (8 pts) Write an ocamllex specification for tokens of the following type:

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

where these represent, respectively, the sequences "+", "-", and any string of one or more characters '0' - '9'. (You will want to use the function `int_of_string: string → int.`)

```

rule tokenize = parse
  '+' { PLUS }
  | '-' { MINUS }
  | ['0'-'9']+ as i { INT (int_of_string i) }
  
```

6. (10 pts) The following grammar has a shift-reduce conflict:

$$S \rightarrow B a \mid b a e$$

$$B \rightarrow b \mid c$$

(a) Show two sentences that lead to the same stack configuration and lookahead symbol, but where the correct parsing action in one case is to shift and in the other is to reduce. Show the two parse trees, and the problematic stack configuration, and say whether to shift or reduce in either case.

**ba and bae lead to stack configurations with b on the stack and a as the lookahead symbol. For ba, the correct action in this configuration is to Reduce using  $B \rightarrow b$ ; for bae the correct action is Shift.**



(b) Give an equivalent grammar – one with the same language – which does not have the conflict.

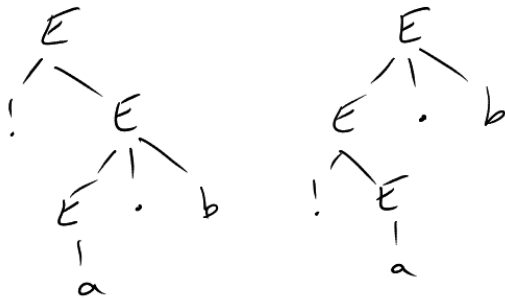
$$S \rightarrow b a \mid c a \mid b a e$$

7. (10 pts) Given the following ambiguous grammar

$$E \rightarrow E . id \mid ! E \mid E [ E ] \mid id$$

a) Give a sentence that has two distinct parse trees. Show the parse trees.

**! a . b (or !a[b], as well as others)**



b) Explain how you might disambiguate the grammar using ocaml yacc precedence and associativity declarations. You are free to choose the precedence order and associativity of operators, but you must say what declarations you would use and what effect they would have. Assume the following declaration:

```
%token DOT IDENT BANG LBRACK RBRACK
```

```
%left DOT
```

```
%nonassoc BANG
```

```
%nonassoc LBRACK
```

**These declarations give precedence to BANG over DOT, so the above sentence would be parsed as the tree on the right. It also gives precedence to LBRACK, which has an ambiguity as noted above.**

8. (10 pts) Suppose we are given grammar that contains the following rules:

```
Expression → "do" Stmt "while" "(" Expression ")"
            | Stmt ";" Expression
            | Ident
```

Assume the token type is

```
type token = DO | WHILE | LPAREN | RPAREN
            | SCOLON | IDENT of string
```

the abstract syntax for the Expression non-terminal is

```
type exp = DoWhile of stmt * exp | Sequence of stmt * exp
         | Id of string
```

and you are provided with the `parseStmt` function of the following type.

```
parseStmt: token list -> stmt * token list
```

Implement the `parseExp: token list -> exp * token list` function. Ignore error cases. Also, assume that DO and IDENT are not in `FIRST(Stmt)`.

**let rec parseExp toks =**

**match toks with**

**DO::rest -> ( match parseStmt rest with**

**(st, WHILE::LPAREN::rest2) ->**

**match parseExp rest2 with**

**(e, RPAREN::etc) -> (DoWhile(st, e), etc) )**

**| IDENT s::rest -> (Id s, rest)**

**| \_ -> ( match parseStmt toks with**

**(st, SCOLON::rest) -> match parseExp rest with**

**(e, etc) -> (Sequence(st, e), etc) )**

For the following questions, use a three-address intermediate representation like the one we used in class. The instructions are:

```

x = y
x = n
x = &y      (get address of variable y)
x = a op b,  where a and b are names or constants, and op is any arithmetic,
              comparison, or boolean operator

JUMP label
CJUMP x, Lt, Lf  (jump to Lt if x is true, Lf otherwise)
x = LOADIND y   (get value in location contained in y)
ERROR          (return an error)

```

You may use any of the following translation schemes, which were discussed in class:

**[S]** = instruction list to compute statement S

**[e]** = pair containing instruction list to compute expression e and variable name giving location of result

**[S]<sub>blab,clab</sub>** = instruction list for S, in context of while statement, where blab is the break label (label after the enclosing while or switch) and clab is the continue label (label to start next iteration of while).

**[S]<sub>blab</sub>** = instruction list for S, in context of break statement, where blab is the break label. (We did not specifically discuss this scheme in class, but it is just a simpler version of the previous scheme.)

**[e]<sub>x</sub>** = instruction list to calculate value of e and place it in x

**[e]<sub>tlab,flab</sub>** = instructions to calculate boolean-valued expression e and jump to tlab if it is true, flab otherwise. (This is called the “short-circuit evaluation” scheme.)

9. (10 pts) FORTRAN used to have an “Arithmetic IF” statement, which jumped to one of three labels depending upon whether the value of an expression was negative, zero, or positive. Suppose we had a similar “structured” statement of the form:

```

CIF (e) {
    NEG S0
    ZERO S1
    POS S2
}

```

(where NEG, ZERO, and POS are new keywords). It would execute S0, S1, or S2, depending upon the value of e, and then jump to the end of the CIF statement (like a regular IF). Furthermore, any of the three statements can contain a `break` statement, which exits from the CIF statement. Give a compilation scheme for CIF.

[CIF (e) { NEG S0 ZERO S1 POS S2 }] =

**let (I, t) = [e]**

**L0, L1, L2, L3, L4 = fresh labels**

**in**

**I**

**t1 = t < 0**

**CJUMP t1, L0, L1**

**L0: [S0]<sub>L4</sub>**

**JUMP L4**

**L1: t1 = t == 0**

**CJUMP t1, L2, L3**

**L2: [S1]<sub>L4</sub>**

**JUMP L4**

**L3: [S2]<sub>L4</sub>**

**L4:**

10. (10 pts) Some languages have a “break-to-label” construct. In these languages, any statement can be labeled, and if a statement labeled L contains a “break L”, it jumps to the end of the labeled statement. For example, in such a language:

while (cond) {		L: while (cond) {
...		...
break;	is equivalent to	break L;
...		...
}		}

and

while (cond) {		while (cond) {
...		L: {
continue;	is equivalent to	...
		break L;
		...
		}
		}

```

...
}

```

To generate code for such a language, you need to handle two new kinds of statements, labeled statements and break-to-label. For various reasons, we cannot assume that the labels that appear in source programs are the same as the labels in IR code, so we still have to generate new labels when generating code. Define a scheme

$[S]_T$  = instruction list

where  $T$  is a table mapping source-level labels to IR labels. You can use the following operations:

```

add: table -> source-label -> IR-label -> table
get: table -> source-label -> IR-label
genlabel: unit -> IR-label.

```

Define  $[L: S]_T$  and  $[\text{break } L]_T$

```

[L: S]T =  let L0 = genlabel()
           in let newT = add T L L0
           in   [S]newT
           L0:

```

```

[break L]T =  let L0 = get T L
           in  JUMP L0

```

11. (10 pts) Fill in the blanks:

- (a) The output of the front-end of a compiler is an AST and a symbol table. The output of the back-end is a machine-language program.
- (b) When a run-time system compiles an intermediate representation to machine language at run time, it is called just-in-time compilation.
- (c) At run time, memory contains the program and static values, and two areas for storing run-time values; these two are the stack and the heap.
- (d) Two languages whose run-time systems provide *automatic* memory management are: OCaml and Java (or C#, Python, Lisp, many others); one language that does not is C (or C++, FORTRAN, many others).
- (e) Name one type of machine-*dependent* optimization: register allocation, or instruction selection, or instruction scheduling.