*Version 1*

CSE 305 Mid-Term Examination

Thursday, March 17, 2022

**11:00 – 12:20 pm**

**Your Full Name:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Student Number:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Instructions:**

1. This is a closed-book test. Switch off all electronic devices.

2. Answer ALL questions, pick only one answer.

3. There are questions that are derived from practices questions but they are different, read carefully!

*Note: Penalty for violation of academic integrity is an F grade on the course, as per CSE Department policy.*

**Grade:**

**Useful definitions:**

**let rec** fold\_left (f:'a->'b->'a) (acc:'a) (l:'b list): 'a =

**match** l **with**

| [] -> acc

| x::xs -> fold\_left f (f acc x) xs

**The type of** fold\_left **is:**

('a->'b->'a) -> 'a -> 'b list -> 'a

**let rec** fold\_right (f:'a->'b->'b) (l:'a list) (acc:'b): 'b =

**match** l **with**

| [] -> acc

| x::xs -> f x (fold\_right f xs acc)

**The type of** fold\_right **is:**

('a->'b->'b)-> 'a list-> 'b -> 'b

**Multiple Choice [Each worth 4] Circle the letter that best answers the question.**

**Consider the following grammar with initial non terminal <Foo>, where** %**,** ^**,** -**,** (**,** )**, and id are terminals for Questions (1) and (2).**

<Foo> ::= <Foo> % <Foo> | <Foo> ^ <Bar>

<Foo> ::= - <Foo> | ( <Foo> )

<Foo> ::= <Bar>

<Bar> ::= <Stool>

<Stool> ::= id

**(1) Pick the answer that best describes the above grammar:**

1. Ambiguous, left recursive
2. Ambiguous, right recursive
3. Unambiguous, left recursive
4. Unambiguous, left and right recursive
5. Unambiguous, right recursive
6. Ambiguous, left and right recursive

**(2) Which of the following sentences are valid in the language defined by the grammar? Assume that any alphabetic character string that contains no white space characters is a valid id:**

1. -(sparkle) % -(twinkle)
2. (sparkle twinkle) ^ -shine
3. –sparkle ^ twinkle ^ (-shine)
4. 3 only
5. 1 & 2
6. 1 only
7. 2 only
8. 1 & 3
9. 1, 2, & 3

**(3) Which statement best describes the below grammar:**

<expr> ::= id | <expr> <expr> + | <expr> <expr> \*

1. it gives right associativity for + and \*
2. it gives left associativity for + and \*
3. it is an ambiguous grammar for a set of postfix expressions
4. it is an unambiguous grammar for a set of postfix expressions
5. it gives right associativity for + and left associativity for \*
6. it gives left associativity for + and right associativity for \*

**Consider the following grammar with initial non terminal** <BoolExp> **for questions (4), (5), and (6). Please observe that the grammar supports both binary and unary operators:**

<BoolExp> ::= id | ( <BoolExp> ) | <UnOp> <BoolExp>

<BoolExp> ::= <BoolExp> <BinOp> <BoolExp>

<UnOp> ::= ^

<BinOp> ::= or | and

**(4) Which set of parse trees shows the grammar is ambiguous?**

**A:**

<BoolExp>

<BoolExp>

id

<BinOp>

or

<BoolExp>

id

<BoolExp>

<BoolExp>

id

<BinOp>

and

<BoolExp>

id

**B:**

<BoolExp>

<BoolExp>

id

<BinOp>

or

<BoolExp>

id

<BoolExp>

<UnOp>

^

<BoolExp>

<BoolExp>

id

<BinOp>

or

<BoolExp>

id

<UnOp>

^

<BoolExp>

<BoolExp>

<UnOp>

id

<BinOp>

or

<BoolExp>

id

<BoolExp>

^

<BoolExp>

<UnOp>

id

<BinOp>

or

<BoolExp>

id

<BoolExp>

^

**C:**

<BoolExp>

<BoolExp>

<BoolExp>

(

or

<BoolExp>

id

<BinOp>

<BoolExp>

(

<BinOp>

or

<BoolExp>

id

<BinOp>

**D:**

<BoolExp>

<BoolExp>

<BoolExp>

<BoolExp>

)

)

id

<BoolExp>

id

<BinOp>

<BoolExp>

<BoolExp>

<BoolExp>

id

or

id

or

**E:** AB C and D all show the grammar is ambiguous

**(5) Rewrite the grammar so that operator “**or**” has right-associativity and operator “**and**” has left-associativity. (you can ignore operator precedence between “**or**” and “**and**”):**

1. <BoolExp> ::= <BoolExp> <BinOp> <Term> | <Term>

<Term> ::= <UnOp> <Factor> | <Factor>

<Factor> ::= ( <BoolExp> ) | id

<UnOp> ::= ^

<BinOp> ::= or | and

1. <BoolExp> ::= <BoolExp> <OrOp> <Term>   
    | <Term> <AndOp> <BoolExp> | <Term>

<Term> ::= <Term> <AndOp> <Factor> | <Factor>

<Factor> ::= <UnOp> <Factor> | <Const>

<Const> ::= ( <BoolExp> ) | id

<UnOp> ::= ^

<AndOp> ::= and

<OrOp> ::= or

1. <BoolExp> ::= <Term> <OrOp> <BoolExp> | <Term>

<Term> ::= <Term> <AddOp> <Term> | <Factor>

<Factor> ::= <UnOp> <Factor> | <Const>

<Const> ::= ( <BoolExp> ) | id

<UnOp> ::= ^

<AndOp> ::= and

<OrOp> ::= or

1. <BoolExp> ::= <Term> <OrOp> <BoolExp> | <Term>

<Term> ::= <Term> <AndOp> <Factor> | <Factor>

<Factor> ::= (<BoolExp>)| <UnOp> <Factor> |<Const>

<Const> ::= id

<UnOp> ::= ^

<AndOp> ::= and

<OrOp> ::= or

1. All of the above (A, B, C, and D)
2. None of the above

**(6) Rewrite the grammar to [1] give precedence to** “and” **over** “or” **for and [2] precedence to unary operations over binary operations:**

1. <BoolExp> ::= <BoolExp> <OrOp> <BoolExp> | <Term>

<Term> ::= ( <BoolExp> ) | <UnOp> <Term>

<Term> ::= <Term> <AndOp> <Term> | id

<UnOp> ::= ^

<AndOp> ::= and

<OrOp> ::= or

1. <BoolExp> ::= <BoolExp> <OrOp> <BoolExp>

| <UnOp> <BoolExp>

<BoolExp> ::= ( <BoolExp> ) | <Term>

<Term> ::= <Term> <AndOp> <Term> | id

<UnOp> ::= ^

<AndOp> ::= and

<OrOp> ::= or

1. <BoolExp> ::= <BoolExp> <AndOp> <BoolExp> | <Term>

<Term> ::= <Term> <OrOp> <Term> | <UnOp> <Term>

<Term> ::= ( <BoolExp> ) | id

<UnOp> ::= ^

<AndOp> ::= and

<OrOp> ::= or

1. <BoolExp> ::= <BoolExp> <OrOp> <Term> | <Term>

<Term> ::= <Term> <AndOp> <Factor> | <Factor>

<Factor> ::= ( <BoolExp> ) | <UnOp> <Factor> |<Const>

<Const> ::= id

<UnOp> ::= ^

<AndOp> ::= and

<OrOp> ::= or

1. <BoolExp> ::= <BoolExp> <AndOp> <Term> | <Term>

<Term> ::= <Term> <OrOp> <Factor> | <Factor>

<Factor> ::= ( <BoolExp> ) | <UnOp> <Factor> | id

<UnOp> ::= ^

<AndOp> ::= and

<OrOp> ::= or

1. A and E

**Consider the following OCaml code for Questions (7), (8), and (9):**

**let** f x = x

**let rec** mystery3(g, n) =

**if** n = 1

**then** g 1

**else** g n + mystery3(g, n-1)

**let** res = mystery3(f, 5)

**(7) What is the type of *f*?**

1. int -> int
2. 'a -> 'b -> 'b
3. 'a -> 'b
4. 'a -> 'a
5. 'a list -> 'a list

**(8) What is the type of *mystery3*?**

1. 'a \* 'a -> 'a
2. ('a -> 'a) \* 'a -> 'a
3. (int -> int) \* int -> int
4. 'a \* int -> int
5. ('a -> 'a) \* int -> int

**(9) What is the value of *res*?**

1. 10
2. 15
3. 54321
4. 14
5. 120

**(10) Consider the following OCaml code:**

type element = Window | Wall | Door | Other

**let** build x room =

**match** x **with**

| Window -> "window"::room

| Wall -> "wall"::room

| Door -> "door"::room

| \_ -> "other"::room

**What is the type of function *build*?**

1. element \* string -> string
2. string \* string -> string list
3. element -> string list -> string list
4. element \* string list -> string list
5. element -> element list
6. element -> string list -> element list

**(11) What is printed to the screen when the following OCaml code is executed?**

**let** g =

**let** s = 4

**in let** f =

**let** s = 3

**in** (print\_endline (string\_of\_int s))

**in let** s = 5

**in** (f; print\_endline (string\_of\_int s))

1. 3 4
2. 3 5
3. 4 3
4. 4 5
5. This code contains a type error

**Consider the following two OCaml functions for questions (12), (13), and (14). Note mod is the modulo function:**

**let** mystery1 l =

List.fold\_left (**fun** acc x -> **if** (x mod 2) = 0

**then** x::acc

**else** acc)

[] l

**let rec** mystery2 l =

**match** l **with**

| x::xs -> **if** (x mod 2) = 0

**then** x::(mystery2 xs)

**else** mystery2 xs

| [] -> []

**(12) Are *mystery1* and *mystery2* equivalent (by equivalent we mean for the same input they produce the same output)?**

1. No, the two functions do not return the same type
2. No, the list produced by mystery1 is the reverse of the list produced by mystery2
3. Yes, both functions return the same lists with the **same elements** in the **same order**
4. Yes, both functions will return the same integer
5. No, the list produced by mystery2 will contain **more** elements than the list produced by mystery1

**(13) What is the type of *mystery2*?**

1. 'a list -> 'a list
2. int list -> int \* int list -> 'a list
3. int list -> int -> int list
4. int list -> int list
5. int list -> int

**(14) What is the type of *mystery1*?**

1. ('a \* 'b -> 'b) -> 'b -> 'a list -> 'b
2. int list -> ('a \* 'b -> 'b) -> 'b -> 'a list -> 'b -> int list
3. 'a list -> ('a \* 'b -> 'b) -> 'b -> 'a list -> 'b -> 'a list
4. int list -> int list
5. 'a list -> 'a list

**(15) Consider the following OCaml code. You can assume this function does not generate any warnings or type errors.**

**let rec** myfun x y z =

**match** x **with**

| B i -> ((y + i), z)

| Q -> (y, (z + 25))

| C (j, k) -> ((y + j), (z + k))

| W (a, b) -> **let** (c, d) = myfun a y z

**in** myfun b c d

**myfun has type:** foo -> int -> int -> int \* int

**How could you define the data type *foo* used in function *myfun*?**

1. type foo = 'a | int | int \* int | foo \* foo
2. type foo = Q | C of int \* int

| B of int | W of foo \* foo

1. type foo = Q | C of int \* int | Empty

| B of int | W of foo \* foo | Err of int

1. type foo = Q | C (int, int)

| B int | W (foo, foo)

1. All of the above are possible
2. None of the above are possible

**(16) Consider the following OCaml code. You can assume this function does not generate any warnings or type errors.**

**let rec** myfun2 x y z =

**match** (x, y) **with**

| (0, \_) -> Empty

| (n, B i) -> B (i \* z \* n)

| (n, W (a, b)) ->

(**match** (myfun2 x a z, myfun2 x b z) **with**

| (Empty, Empty) -> Empty

| (B i, B j) -> B (i + j)

| \_ -> Empty)

| \_ -> Empty

**myfun2 has type:** int -> ty -> int -> ty

**How could you define the data type *ty* used in function *myfun2*?**

1. type ty = W of ty \* ty | B of int | Empty   
    | Err of int
2. type ty = C of int \* int | B of int | W of ty \* ty

| Empty | Err of int

1. type ty = P | N | D | Q | C of int \* int

| B of int | W of ty \* ty | Empty

1. type ty = W of ty \* ty | B of int | Empty
2. None of the above are possible
3. All of the above are possible

**(17) What is printed to the screen when the following OCaml code is executed?**

**let** x = 1

**in let** y = 2

**in let** f y = (print\_endline(string\_of\_int(x+y)); y)

**in let** x = 3

**in let** y = 4

**in let** g y = print\_endline(string\_of\_int(x+y))

**in let** x = 5

**in** g (f 3)

1. 6 8
2. 4 6
3. 3 7
4. 4 8
5. This code contains a type error

**(18) Consider the following functions in OCaml:**

**let** foo(x) = (print\_endline("foo");x)

**let** bar(x) = (print\_endline("bar");x)

**let** baz(x) = (print\_endline("baz");x)

**Which of the following statements will print the same thing to the screen:**

1: **let** \_ = foo baz bar 1

2: **let** \_ = foo (bar (baz 1))

3: **let** \_ = foo bar (baz(1))

1. 1, 2, and 3 will print the same thing
2. 2 and 3 will print the same thing
3. 1 and 2 will print the same thing
4. 1 and 3 will print the same thing
5. None, they all print something different

**(19) Given a function**

**let** helper a =

**match** a **with**

| Some x -> x

| None -> false

**Which of the following uses of *helper* is correct (does not produce a type error):**

1. helper (Some true) || helper (None)
2. helper (Some true || None)
3. Some (helper true) || Some (helper false)
4. None of the above
5. All of the above

**(20) Consider the following type definition and function in OCaml :**

**type** markup = Ital | Bold | Font of string

**let** chmarkup (m:markup) : markup =

**match** m **with**

| Font "Courier" -> Font "Arial"

| Font "Arial" -> Font "Courier"

| \_ -> m

**let rec** map f xs =

**match** xs **with**

| [] -> []

| hd::tl -> (f hd)::(map f tl)

**What is the result of the following function call?**

map chmarkup [Font "Courier"; Font "Arial"; Ital;

Font "Times New"; Font "Arial"; Bold]

1. [Font "Arial"; Font "Courier"; m;

m; Font "Courier"; m]

1. [Font " Arial"; Font "Courier"; Ital;

Font "Times New"; Font "Courier"; Bold]

1. [Font "Courier"; Font "Arial"; Ital;

Font "Times New"; Font "Arial"; Bold]

1. This code has a Type Error
2. None of the Above

**(21) Given that the function *filter* is defined as:**

**let rec** filter f xs =

**matc**h xs **with**

[] -> []

| hd :: tl -> **if** f hd

**then** hd :: (filter f tl)

**else** filter f tl

**Type of *filter* is:** (’a -> bool) -> ’a list -> ’a list

**and the *reduce* function seen in class is defined as:**

**let rec** reduce f u xs =

**match** xs **with**

| [] -> u

| hd :: tl -> f hd (reduce f u tl)

**Type of *reduce* is:** ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b

**How would you define *filter* in terms of *reduce*?**

**A.** let filter f xs = reduce (**fun** f a -> **if** f a **then** a **else** f) [] xs

**B.** let filter f xs = reduce (**fun** a b -> **if** f a **then**

a :: xs **else** xs) [] xs

**C.** let filter f xs = reduce f [] xs

**D.** let filter f xs = reduce (**fun** a b -> **if** f a **then**

a::b **else** b) [] xs

**E.** filtercannot be defined in terms ofreduce

**(22) Which answer best describes the following statement: Every function in OCaml takes one argument.**

**A :** The statement is True

**B :** The statement is False