SC-T-501-FMAL Programming languages
Final exam

Spring 2020

*** SOLUTIONS ***

Teacher: Tarmo Uustalu
Date and time: Thursday, 16 April 2020, 14:00-17:00

Write answers in English or Icelandic.

Write the answers in the plaintext template file provided. Upload the file through Canvas by 17.00.

(If you are in the special needs group, then by 18.00.)

Should any issues arise, you can also send your solution to tarmo@ru.is.

You can ask questions by Skype (chat) to tarmo.uustalu (this is strongly preferred). Or by email to tarmo@ru.is (less preferred).

Best of luck!

Your name: ______________________________

Kennitala: ______________________________

Part 1: Multiple-choice questions

Mark your chosen answer by putting an asterisk (*) at the beginning of the line with the correct answer. Only mark one answer.

1. (2 p) Statements (also called commands)

   a. return a value, do not normally produce a side-effects (read or write the store)
   [✓ b.] return “nothing”, usually produce a side-effect

2. (2 p) List types collect

   a. fixed-lengths sequences of values with a different type for every position in the sequence,
   [✓ b.] arbitrary-length sequences of values of the same type

3. (2 p) The unit type contains

   a. no values
   [✓ b.] one value
   c. infinitely many values

4. (2 p) The tag `Some` of the discriminated union type `option` signifies

   a. undefinedness (failure)
   [✓ b.] definedness (success)

5. (2 p) Prettyprinting means

   [✓ a.] converting abstract syntax into concrete syntax
   b. converting concrete syntax into abstract syntax

6. (2 p) The result of successful lexing is

   a. a string
   [✓ b.] a list of tokens
   c. an abstract syntax tree

7. (2 p) An environment is a dictionary that has variable names as

   [✓ a.] keys
   b. values

8. (2 p) Shadowing means that defining a variable in a block

   [✓ a.] makes non-local variables with the same name inaccessible inside this block
   b. makes non-local variables with the same type inaccessible inside this block

9. (2 p) Compilation means
   a. “direct” execution of programs of a language.
   [✓ b.] translation of programs in one language to another.

10. (2 p) Running the compiled form of an expression on a stack machine

    [✓ a.] grows the stack height by one
    b. does not change the stack height
    c. shrinks the stack height by one

11. (2 p) That a program may divide by zero on some inputs

    a. is usually detected by the language's type system
    [✓ b.] usually only manifests when the program (or its compiled form) is run

12. (2 p) Nearly all modern languages use

    a. the dynamic scope rule
    [✓ b.] the static scope rule

    (for non-local variables in function definitions)

13. (2 p) A language being higher-order means that

    a. functions can be defined locally
    [✓ b.] functions are values that can be passed as input to other functions
    c. functions can be recursive

    (Mark only the criterion that any higher-order functional language must meet.)

14. (2 p) The term-forms of lambda-calculus are: variables, applications and

    a. alpha-conversions
    b. beta-reductions
    [✓ c.] lambda-abstractions

15. (2 p) Variable capture means that

    [✓ a.] some free variables become bound that should not
    b. some bound variable become free that should not

16. (2 p) Given an environment and store, an access expression has

    [✓ a.] both an l-value and an r-value
    b. only an l-value
    c. only an r-value

17. (2 p) If `x` has been declared as an integer, then the l-value of `x` is

    [✓ a.] an address
    b. an integer

18. (2 p) If `x` has been declared as an integer, then the r-value of `&x` is

    [✓ a.] an address
    b. an integer

19. (2 p) In reference-counting based garbage collection, an object may be put on the freelist when its reference counter has become

    a. 1
    [✓ b.] 0
    c. 2

20. (2 p) In the sweep phase of mark-and-sweep garbage collection

    a. all marked blocks are put on the freelist
    [✓ b.] all unmarked blocks are put on the freelist

Part 2: Freetext questions

In Problems 21-23, it is more important to demonstrate that you understand how to solve the problem than to resolve any errors that F# may give because you have got something wrong in the syntax.

21. (6 p) Write an F# function `drop23 : 'a list -> 'a list` that drops the 2nd and 3rd element of a given list and leaves the rest of the list intact. If the list does not have a 2nd or 3rd element, don't do anything about them, just keep the other elements.

```
> drop23 [10; 11; 12; 13; 14; 15];;
val it : int list = [10; 13; 14; 15]

> drop23 [10; 11];;
val it : int list = [10]

> drop23 [10];;
val it : int list = [10]
```

```
let drop23 xs =
    match xs with
    | [] -> []
    | [x1] -> [x1]
    | [x1;x2] -> [x1]
    | x1::x2::x3::xs -> x1::xs
```

One can also group the cases better:

```
    | [x1;x2] -> [x1]
    | x1::x2::x3::xs -> x1::xs
    | xs -> xs
```

22. (6 p) Using `List.filter` and `List.map`, write a function `sqMul5 : int list -> int list` that returns the squares of those elements of a given list that are multiples of 5.

```
> sqMul5 [1; 3; 5; 8; 10; 15; 7];;
val it : int list = [25; 100; 225]
```

```
let sqMul5 xs = List.map (fun x -> x * x) (List.filter (fun x -> x mod 5 = 0) xs)
```

23. (6 p) Using recursion and pattern-matching on the list (and in a relevant case also on the result of a recursive call), write a function `safeTake : int -> 'a list -> ('a list) option` that returns the given number of first elements of a given list if the number is non-negative and the list has many enough elements. The function should signal failure if the number is negative or the list is too short.

```
> safeTake 3 [16; 20; 5; 4; 2];;
val it : (int list) option = Some [16; 20; 5]

> safeTake 2 [5];;
val it : (int list) option = None
```

```
let rec safeTake n xs =
    match n, xs with
    | 0, _  -> Some []
    | _, [] -> None
    | n, x::xs -> match safeTake (n-1) xs with
                  | None    -> None
                  | Some ys -> Some (x :: ys)
```

With a library functiton, the last case can more compactly be coded like this:

```
    | n, x::xs -> Option.map ((::) x) (safeTake (n-1) xs)
```

24. (4 p) Suppose that // and ∗∗ are two binary infix operators whereby // binds stronger than ∗∗, that // is left-associative and ∗∗ is right-associative.

    Disambiguate the following expression for a reader that does not know about these conventions by parenthesizing all subexpressions of the following expression, variable names and the whole expression excepted:

    $x//y//z//w**a//b**c$

    $(((x//y)//z)//w)**((a//b)**c)$

25. (6 p) Assume that concrete syntax of expressions is given by the grammar

```
e --->                        // expressions
    | s ss

ss --->                       // lists of summands
    | + s ss | - s ss
    | (empty)

s --->                        // summands
    | f ff

ff --->                       // lists of factors
    | * f ff
    | (empty)

f --->                        // factors
    | x
    | i
    | (e)

i                             // integer tokens

x                             // variable name tokens

+ - * ( )                     // other tokens
```

Consider this string:

`c + (a + 78) * ((b))`

To parse this string as an expression, which substrings need to be parsed

(i) as expressions (category e)?

`c + (a + 78) * ((b))`, `a + 78`, `(b)`, `b`

(ii) as summands (category s)?

`c`, `(a + 78) * ((b))`, `a`, `78`, `(b)`, `b`

(iii) as factors (category f)?

`c`, `(a + 78)`, `a`, `78`, `((b))`, `(b)` `b`

26. (6 p) What is the value of the following expression of the first-order functional language?

```
let x = 17 in
  let f y = x + y in
    let y = 3 in
       let x = 42 in
         f (y * 2)
```

(i) under the static scope rule

17 + 3 * 2 = 23

(ii) under the dynamic scope rule

42 + 3 * 2 = 48

27. (4 p) What is the value of the following F# expression? `eval` is the evaluator for the higher-order functional language.

```
eval (Call (Var "f", Mult (Var "x", CstI 2)))
      ["x", 15; "g", 7;  "f", F ("g", "y", Add (Var "y", Var "z"), ["z", 9]), ...]
```

The ellipsis ... can stand for some further variable-value pairs that you need not care about.

39

Explain.

The function `f` is called on $15 * 2 = 30$, resulting in $30 + 9 = 9$.

28. (6 p) In problem 27, how can it be that the closure that is the value of the function `f` has `g` (rather than `f`) recorded as the function's name?

It may be that a function `g` was defined with `LetFun`, giving rise to the closure, and then a function `f` was defined with `Let` as `g`.

Explain how such a situation can arise. If you can, write a nesting of let-blocks

```
let .. = .. in
  let .. = .. in
    let .. = .. in
      let .. = .. in
        let .. =  .. in
          f (x * 2)
```

where the definitions set up the environment as above.

```
let z = 9 in
  let g y  = y + z in
    let f = g in
      let g = 7 in
        let x = 15 in
          f (x * 2)
```

29. (3 p) Still about Problem 27:

Why are we recording not just the parameter's name but also the function's name in a closure anyway?

Since the function's name may occur in its body.

For which kind of functions is it relevant to record the function's name in the closure?

For recursive functions.

Was it relevant in the previous problem?

No.

30. (3 p) Calculate the result of this lambda-term substitution:

$$(x\,(\lambda x.\,x\,(y\,x)))[z\,w/x]$$

(Pay attention to which occurrences of x in the lambda-term to substitute into are free and which are bound.)

$z\,w\,(\lambda x.\,x\,(y\,x))$

31. (4 p) Normalize this lambda-term (i.e., beta-reduce it until you reach a beta-normal form) using the leftmost-innermost reduction strategy.

$$(\lambda x.\, y\,((\lambda z.\, x) w))\,(y\, v)$$

(Recall that the body of a lambda-abstraction spans to the nearest closing parenthesis.)

$$\begin{array}{ll} & (\lambda x.\, y\,((\lambda z.\, x) w))\,(y\, v) \\ \to & (\lambda x.\, y\, x)\,(y\, v) \\ \to & y\,(y\, v) \end{array}$$

32. (6 p) What will the following program print?

```
int* p;
int* q;
int x;
x = 3;
p = &x;
print *p;
q = p;
*p = *q * 5;
print x;
```

3 and 15

Here is why:

```
int* p;
int* q;
int x;
x = 3;              // x is set to 3
p = &x;             // p is set to address of x
print *p;           // content of p, i.e., 3, is printed
q = p;              // q is set to p, so both are address of x
*p = *q * 5;        // content of p is set to (content of q) * 5,
                    // i.e., x is set to 3 * 5 = 15
print x;            // 15 is printed
```