**Name, SURNAME and ID ⇒**

Middle East Technical University — Department of Computer Engineering

# CENG 242

## Programming Language Concepts

Spring '2014-2015

## Midterm Exam

- **Duration: 120** minutes.
- **Total Points: 100**
- **Exam:**
  - This is a **closed book**, **closed notes** exam. The use of any reference material is strictly forbidden.
  - No attempts of cheating will be tolerated.
- **This exam consists of 7 pages including this page. Check that you have them all!**
- **GOOD LUCK !**

| | |
|---|---|
| Question 1 | |
| Question 2 | |
| Question 3 | |
| Question 4 | |
| Question 5 | |
| Question 6 | |
| Total ⇒ | |

## QUESTION 1. (18 points)

Given the following program written in a C-like language:

```
int a = 3;

void f(int b) {
    int c = -1;
    static int f = 7;

    if (b > 0) {
        int d = 17;
        printf("%d\n", d); // POINT 1
    }
    else {
        int e = -6;
        printf("%d\n", e); // POINT 2
    }
    ++f;
    int* p = (int*) malloc(3 * sizeof(int));
    p[0] = 1 + b; p[1] = 2 + b; p[2] = 3 + b;
}

int main() {
    f(5); f(-5);
    printf("Done\n"); // POINT 3
    return 0;
}
```

Write the values of the variables stored in global, stack (local variables), and heap (dynamic memory allocations) memory regions for the above program for each specified point in the program. Each box may contain at most one variable's value (or it can be empty). Assume that memory for each region is given from the first available location. Fill the boxes in bottom-to-top order. Do not write variable names; write their values in appropriate boxes.

| POINT 1 | | | POINT 2 | | | POINT 3 | | |
|---|---|---|---|---|---|---|---|---|
| Global | Stack | Heap | Global | Stack | Heap | Global | Stack | Heap |
| | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | -2 |
| | | | | | | | | -3 |
| | | | | | | | | -4 |
| | 17 | | | -6 | 8 | | | 8 |
| 7 | -1 | | 8 | -1 | 7 | 9 | | 7 |
| 3 | 5 | | 3 | -5 | 6 | 3 | | 6 |

## QUESTION 2. (20 points)

**a)** Write the output of the following program for each parameter passing mechanisms assuming eager evaluation. For the copy-out case, assume that all parameters are initialized to zero. Note that some boxes that correspond to some printf statements are crossed-out. Fill-out only empty boxes.

```
int a = 1, b = 7, arr[3] = {0, 1, 2};
void f(int x, int i) {
    i += a;
    printf("%d ", x);
    x = x - b; a++;
    printf("%d ", x);
}
int main() {
    f(arr[a], a);
    printf("%d ", a);
    for(int i = 0; i < 3; ++i) printf("%d ", arr[i]);
    return 0;
}
```

| | | | | | | |
|---|---|---|---|---|---|---|
| **Copy-in:** | 1 | -6 | 2 | X | 1 | X |
| **Copy-in-out:** | 1 | -6 | 2 | X | -6 | X |
| **Copy-out:** | 0 | -7 | 1 | X | -7 | X |
| **Reference:** | 1 | -6 | 3 | X | -6 | X |

**b)** Answer the following as TRUE or FALSE.

- Assuming that a language is side-effect free, eager evaluation, lazy evaluation, and normal order evaluation all give the same results for all programs written in that language: False, because some expressions may not be evaluated at all.
- In lazy evaluation, expressions are evaluated on a per-need basis. In other words, an expression may not be evaluated at all if its result is not required: True
- Reference and copy-in-out mechanisms always yield the same result although the former is more efficient: False, especially if the argument is a global variable and the function modifies that variable.
- Garbage is a term that is used to represent a pointer variable whose memory is destroyed: False, it is dangling reference.

## QUESTION 3. (16 points)

Mixing lazy and normal-order-evaluation with side effects can lead to unexpected results. With this information in mind, write the output of the following program for both static and dynamic binding. For each binding, write the output of lazy and normal-order evaluation orders:

```
int j = 2, lost[6] = {4, 8, 15, 16, 23, 42};

void f(int v) {
    j = j + 1; printf("%d ", j); printf("%d ", v);
    j = j + 2; printf("%d ", j); printf("%d ", v);
}

int main() {
    int j = 0;
    f(lost[j]);
    return 0;
}
```

| | |
|---|---|
| **Lazy evaluation (static binding):** | 3 16 5 16 |
| **Normal-order evaluation (static binding):** | 3 16 5 42 |
| **Lazy evaluation (dynamic binding):** | 1 8 3 8 |
| **Normal-order evaluation (dynamic binding):** | 1 8 3 16 |

## QUESTION 4. (15 points)

**a)** Assume that a two dimensional matrix type in Haskell is defined as follows:

```
data Matrix a = Matrix [[a]] deriving Show
```

For example, `Matrix [[1,2,3],[4,5,6]]` defines a 2×3 matrix with 2 rows and 3 columns. Implement the following one-liner functions that return the number of columns, the contents of a given row and column in a list, and the sum of all elements. You can use the built-in Haskell functions such as **head**, **tail**, **length**, and **sum**. Do not use other built-in functions. Do not define extra functions or write outside of the boxes (violations will automatically get 0 credit). Hint: consider using list comprehensions.

numCols (Matrix a) = $\boxed{\text{length (a!!0)}}$

getRow (Matrix a) i = $\boxed{\text{a!!i}}$

getCol (Matrix a) j = $\boxed{\text{[row!!j | row <- a]}}$

sumAll (Matrix a) = $\boxed{\text{sum [sum row | row <- a]}}$

**b)** Write the type of the following function in standard Haskell notation and using lowercase letters for type variables (Hint: it is polymorphic).

```
examFunc f g s [] = g s
examFunc f g s (a:b) = g (f a (examFunc f g s b))
```

examFunc :: $\boxed{\text{(t -> t2 -> t1) -> (t1 -> t2) -> t1 -> [t] -> t2}}$

## QUESTION 5. (15 points)

Assume you are asked to define a Haskell data type for storing structure of a simple functions of a simple programming language. Your language is defined as follows:

- An expression can be either of:
  - a simple integer value
  - a variable with a string name.
  - a definition with a variable name and an expression
  - a binary operator with two opperand expressions
  - a block expression consisting of **one or more** expressions
- A function consists of a function name and a block expression

**a)** Write a Haskell data type definition FunctionType for representing a Function as defined above. You can define any other data types as needed, but try to keep it minimum. You can use existing Haskell types like `String` whenever convenient.

```
data Expression = Simple Integer | Var String | Definition String Expression |
                  Binop Expression Expression | Block BlockExpr
data BlockExpr = One Expression | More Expression BlockExpr
data FunctionType = Function String BlockExpr
```

**b)** Write a function countdefs :: FunctionType -> `Integer` that will count number of definitions in a function.

```
countdefs (Function _ a) = cd2 a where
        cd2 (One expr) = cd3 expr
        cd2 (More expr more) = cd3 expr + cd2 more
        cd3 (Definition _ expr) = 1 + cd3 expr
        cd3 (Binop expr1 expr2) = cd3 expr1 + cd3 expr2
        cd3 (Block bexpr) = cd2 bexpr
        cd3 _ = 0
```

**c)** What to we need to change in order to make this type an Abstract Data Type (answer in a single sentence)

Hide all data constructors but FunctionType with a `module(...) where` clause

## QUESTION 6. (16 points)

Assume following Haskell definition is given:

```
pfmr [] s = s
pfmr (f : frest) s = pfmr frest (f s)
```

**a)** What is the value of `pfmr [(+ 3), (* 2), (+ 1)] 1`:

$\boxed{9}$

**b)** What is the most general type (inferred type by Haskell) for `pfmr`? (Hint: it is polymorphic)

$\boxed{[\alpha \rightarrow \alpha] \rightarrow \alpha \rightarrow \alpha}$

**c)** Show evaluation steps of expression `pfmr [(+ 1), (* 2)] 1` assuming Haskell uses **Eager evaluation**.

`pfmr [(+ 1), (* 2)] 1` $\mapsto$ `pfmr [(* 2)] ((+ 1) 1)` $\mapsto$

`pfmr [(* 2)] 2` $\mapsto$ `pfmr [] ((* 2) 2)` $\mapsto$ `pfmr [] 4` $\mapsto$ 4

**d)** Show evaluation steps of expression `pfmr [(+ 1), (* 2)] 1` assuming Haskell uses **normal order evaluation**.

`pfmr [(+ 1), (* 2)] 1` $\mapsto$ `pfmr [(* 2)] ((+ 1) 1)` $\mapsto$

`pfmr [] ((* 2) ((+ 1) 1))` $\mapsto$ `((* 2) ((+ 1) 1))` $\mapsto$ `((* 2) 2)` $\mapsto$ 4