Advanced Programming

900294SCIY

MOCK Exam 2 SOLUTIONS

—,
Duration of the exam is 90 minutes

The exam was made by Lorenzo Galeotti & Breanndán Ó Nualláin and peer-reviewed by Lorenzo Galeotti & Breanndán Ó Nualláin

Student name_____

Teacher name_____

Amsterdam University College
Advanced Programming

Instruction:

Please ensure the following:

- You are allowed to use a blue or black pen, but nothing else.
- Write your name on the front page before you answer the questions.
- Do not remove the staple from the exam and keep all sheets connected!
- Write your answers on this exam, in the empty space after the question.
- Switch all electronic devices to silent and put them in your bag. Your bag and other belongings should be placed against the front wall of the classroom. You may have only a pen and a water bottle at your desk.
- Any violation of AUC's rules on fraud may lead to sanctions, ultimately to the exclusion of all examinations for one year (AS&P appendix 2, Regulations governing fraud and plagiarism).
- The number of points per question is specified in the table below.

The result of this test has a weight of 28% in your final grade. The points of the individual questions are:

Part I: 10 + 10 + 10= 30;
Part II: 15 + 10 = 25 ;
Part III: 15 + 30 = 45 .

Your exam grade will be: (points received)/10

Part I: Physical Level & Assembly

Question 1 (10 pts.) In this exercise you should assume that the only physical gate provided to you are NOT and AND.

Consider the following truth table:

a	b	c	$F(a,b,c)$
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Use AND and NOT to implement a chip that computes the boolean function F described by the table. You can describe your chip using: the NAND to tetris Hardware Description Language, propositional logic formulas, drawings (in this case write names on your chips).

Answer:

We will use propositional logic to solve this exercise. First note that disjunction can be easily be written in terms of conjunction and negation:

$$\varphi \vee \psi \equiv \neg(\neg\varphi \wedge \neg\psi).$$

So, in the rest we will freely use conjunction, negation, and disjunction.

We also note that conjunction and disjunctions are associative, so we can write $\varphi \wedge \psi \wedge \chi$ and $\varphi \vee \psi \vee \chi$ without ambiguity.

Using the standard naive algorithm to transform truth table into formulas we get:

$$F(a,b,c) \equiv (\neg a \wedge \neg b \wedge \neg c) \vee (\neg a \wedge b \wedge \neg c) \vee (\neg a \wedge b \wedge c) \vee (a \wedge b \wedge c) \equiv (\neg a \wedge \neg b \wedge \neg c) \vee (\neg a \wedge b \wedge \neg c) \vee (b \wedge c)$$

where the last equivalence is obtained using basic logic transformations.

Question 2 (10 pts.) Consider the following pseudo-code:

```
RAM[0]=RAM[1]+124
```

Write down the corresponding assembly program written in the Hack Assembly Language. Explain your code.

Answer:

Recall that in Hack Assembly the only possible constant that can appear to the left of an assignment is 1. So we will first use the address register A to store in D the number 124.

```
@124
D = A
```

Then we add the (virtual) register 1 to D. We first insert 1 (the address of RAM[1]) into A and then read from memory. So, the resulting code is the following:

```
@1
D = D + M
```

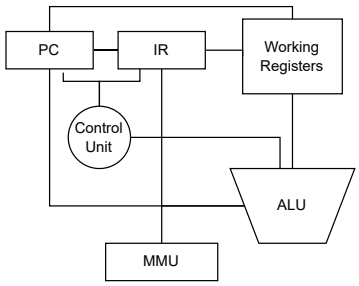
Finally we update $RAM[0]$. As before we first set A to 0 and then use M:

```
@0
M = D
```

Question 3 (10 pts.) Give a short description of a typical CPU. Describe the main components, and what's their role in the interpretation of the machine code. You can use a drawing to help your explanation.

Answer:

This is a typical schema of a CPU:



The main components shown in the figure are:

- PC: Program counter a register that contains the address of the instruction to be executed.
- IR: Instruction register, a register that contains the instructions that needs to be executed.
- Working registers: This is typically a small set of registers that can be used to do calculations.
- Control unit: This is a chip that guides the interpretation using the functionalities provided by the other units.
- MMU: Memory Management Unit, this unit provides an interface with the main memory.
- ALU: Arithmetic Logic Unit, this unit implements the basic arithmetical and logical operations that can be performed on registers.

A typical interpretation cycle in such a machine goes as follows:

- **Fetch:** The instruction at the address stored in the program counter (PC) is loaded in the instruction register (IR) and PC is incremented.
- **Decode/Data Fetch:** The Control Unit decodes the instruction, fetches the needed data (using the MMU if needed), and selects the correct operation to execute (using parameters of ALU and/or MMU).
- **Execution:** The operation is then executed and the result stored in the appropriate address.

Part II: Theory of Languages

Question 4 (15 pts.) Answer **one** of the following two questions:

1. Explain the main differences between pure interpretation and pure compilation of computer languages. List some pros and cons of the two approaches.

2. Explain the difference between eager and lazy evaluation strategies. Mention pros and cons of the two approaches. Then give an example of a program whose behaviour changes depending on whether the evaluation strategy is eager or lazy, explain your example.

Answer:

1. We consider the setting in which we have a host machine which is capable of executing code written in L_{Host} and assume that we want to execute programs written in a different language L_{Target}.

   - **Pure Interpretation:** In this case we write an interpreter of L_{Target} in L_{Host}, i.e., an L_{Host} program that given an L_{Target} program executes it line by line. Interpreters are often easier to write but are also usually quite slow, indeed, the time needed for the translation is part of the execution time of the program to be executed.
   - **Pure Compilation:** In this case we write an L_{Host}-program that translates any code written in L_{Target} to a program written in L_{Host}, i.e., it generates an equivalent program written in L_{Host}. Compilers are usually quite efficient since they allow for code optimization and remove the additional interpretation cycle needed for interpreters. On the other hand pure compilation may be really unfeasible if target and host languages are "too far" from each other, indeed, in this case pure compilation may lead to explosion of code size.

2. When evaluation expressions in programming languages semantics there are two main approaches:

   - Eager: Before evaluating an expression we first evaluate all its sub-expressions.
   - Lazy: When evaluation an expression we do not need to evaluate all the sub-expressions first, but we only evaluate expressions when it is strictly needed for the calculation.

   Typically eager is easier to implement given that the same policy applies to all operators while lazy requires ad-hoc policies for each operator. On the other hand lazy can be used to implement both strict and non strict operators while in general eager cannot.

   Consider the following program:

   ```
   1 == 1 and 1 == 1/0
   ```

   the previous program will raise an exception if the evaluation strategy for the and operator is eager (and is implemented as strict), while will output be evaluated to true if the strategy is lazy (and is implemented as non-strict).

Question 5 (10 pts.) Explain the difference between abstract and concrete syntax. In particular, say how and why abstract syntax is used. Then give an example that shows a typical use the abstract syntax.

Answer:

We assume that we are working in a language L. The concrete syntax of L is the syntax used by the programmer to write programs in L. The abstract syntax is the syntax used by the interpreter/compiler to represent programs written in L. Typically the abstract syntax contains more information than the concrete syntax. This information is mainly there to facilitate the translation and to remove apparent ambiguities.

A typical example is a language in which the programmer can write expressions like this:

$$1 + 2 * 3$$

without a given order of the operators this expression is ambiguous. To solve this in abstract syntax (to include order of operators in the syntax) we typically use abstract syntax trees (often in the form of Polish notation expressions) which disambiguate expressions of this type, e.g., the previous expression would be parsed to $(+1(*23))$ if the usual order of arithmetical operations is adopted. Another paradigmatic example of apparent ambiguity is assignment:

$$x = x + 1$$

where the two occurrences of x have different meaning. This is typically solved by labeling the two occurrences in a different way, e.g., we can explicitly label the first occurrence as the location *denoted* (Den("x")) by "x" and the second as the *value* of the location denoted by "x" (Val(Den("x"))).

Part III: Interpreters

Question 6 (15 pts.) Explain how parameter passing by-name works. Then explain how this can be simulated in a language with higher-order functions.

Answer:

In pass by-name we want to pass to the subprogram an expression that is not evaluated during parameter passing.

So, in this type of parameter passing, actual parameters will be syntactic expressions and formal parameters will be names. During parameter passing, the actual parameter (the syntactic expression) is bound to the formal parameter.

Then, inside the subprgram we evaluate the expressions associated to the formal parameters only when they are needed. Finally, note that the expression may contain names, these names are usually evaluated in the environment where the function was called.

While hour languages do not implement this mode, we can easily simulate parameter passing by-name by passing an anonymous function which has no parameters.

Question 7 (30 pts.) Consider a language with the following abstract syntax in Lisp:

```
(defunion Exp
  (int (i integer))
  (bool (b boolean))
  (mult (l Exp) (r Exp))
  (equal (l Exp) (r Exp))
  (lambda (parameters (list-of symbol)) (body Exp))
  (apply (fun Exp) (arguments (list-of Exp)))
  (address (i symbol))
  (var (v symbol)))

(defunion Com
  (assign (v symbol) (val Exp))
  (if-c (test Exp) (then Com) (else Com)))
```

The intended semantics of these construct is the same as what we saw during the course. Implement an interpreter for this language in Python. Comment on the choices you made while doing so, e.g., scoping rule, parameter passing technique etc.

Answer:

LISP SOLUTIONS HERE