ITI1120 – Introduction to Computing I
Exercise Workbook
Fall 2011

Table of Contents

1. Introduction

This work book contains the exercises that shall be studied in class. Please bring it to all classes. The front matter (sections 1 to 4) of the workbook provides some background information that can be useful in completing the exercises as well as help in completing assignments. It includes:

- **Computer Programming Model** – The programming model is a representation of the main computer components that are used to execute programs. The model is useful in understanding how a program is executed. It is used with most of the exercises in the work book. The model is briefly described in Section 2. This section also gives two blank programming model pages that can be used for your studying and assignments.
- **Pre-defined Algorithm Models** – A number of pre-defined "standard" algorithm models are available for developing algorithms. They represent standard functions available in a computer system such as reading from the keyboard and writing to the terminal console window.
- **Program Structure, Algorithm and Java Programming Concepts** – This section provides a table that summarizes many of the programming structure and concepts studied in class and during the labs. The table can serve as a quick reference for the important representation of algorithm concepts and Java programming concepts/details.

2. Computer Programming Model

The two main components used in executing a computer program are the computer's main memory and the central processing unit (CPU). The computer memory contains the program in the form of machine instructions – these machine instructions are typically created from higher level languages such as C, Java (using compilers – more on this in class).

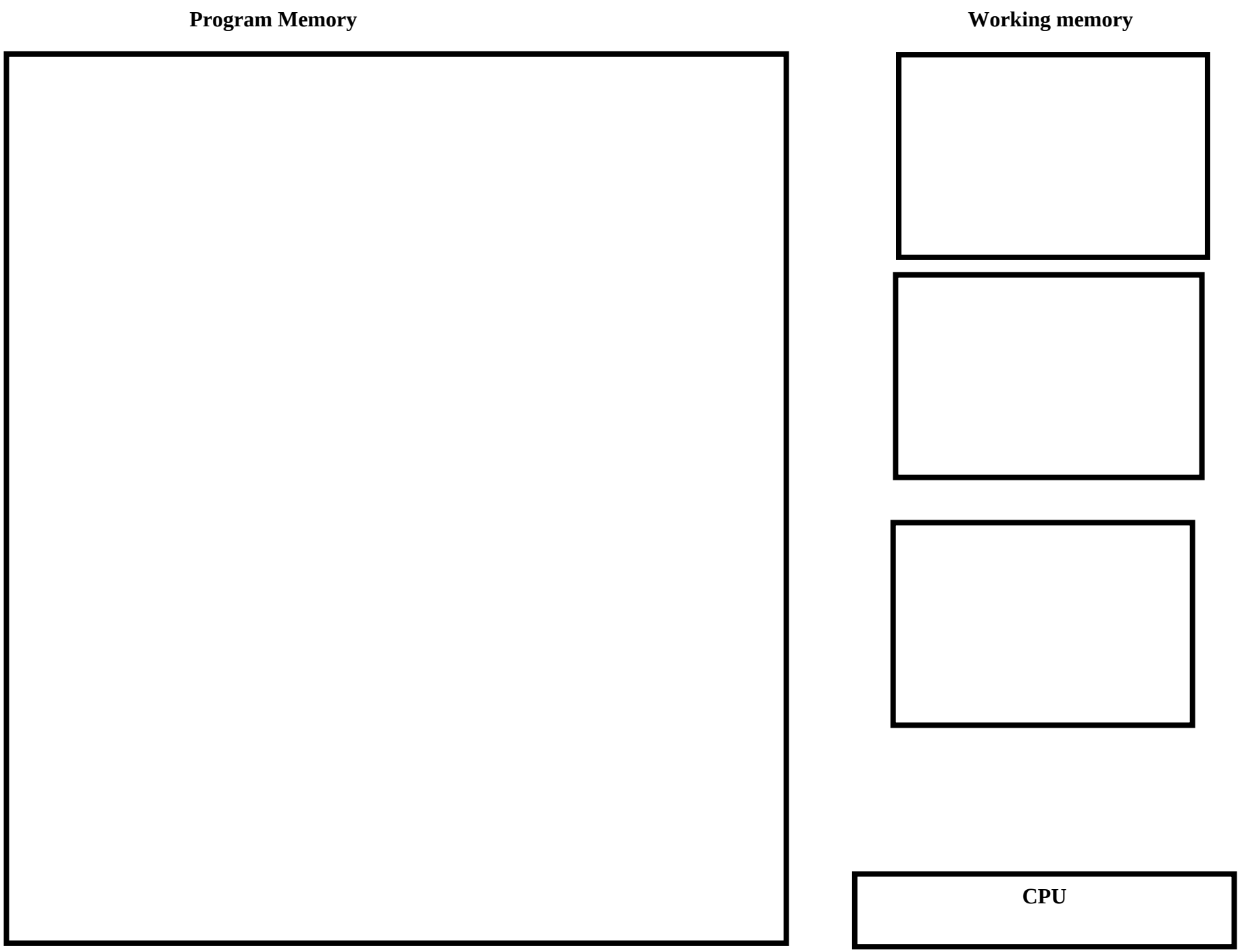
The machine instructions operate on variables, which are also located in the computers memory. Essentially the CPU reads values from and writes values to the variables (i.e the locations in memory where variables are located). Operations such as addition, incrementing, testing, are all typically done within the CPU.

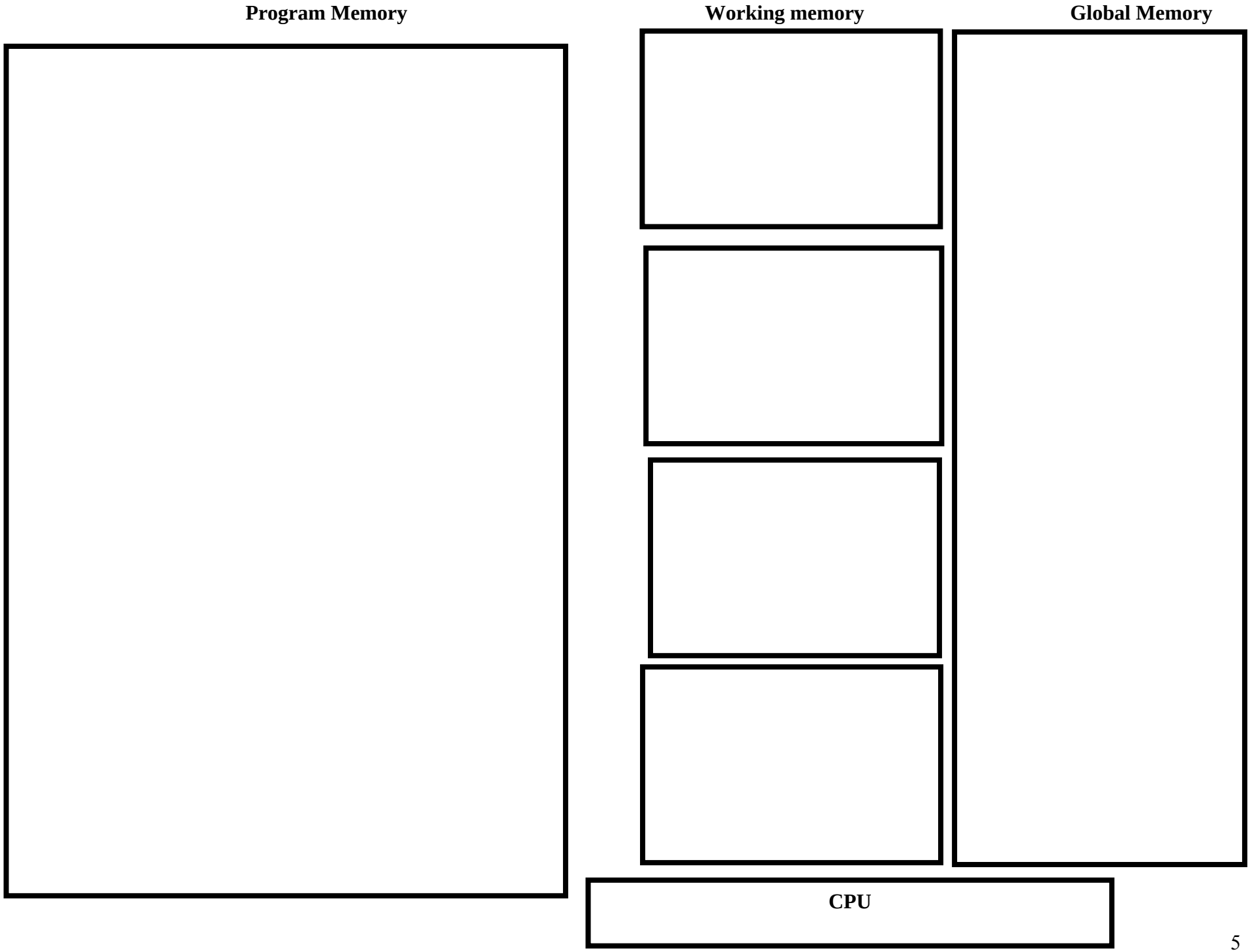
Computer memory consist of a sequence of bytes (8 bit values; a bit is a digit that can assume the value 0 or 1 and a byte contains 8 bits). Thus a computer that contains 3 GigaBytes of memory contains 3 x 1024 x 1024 x 1024 bytes, that is, over 3 billion bytes.

Each of these bytes has an address. An address is simply a number that points to each byte in memory. Thus when the CPU wants to execute an instruction it reads the instruction using its address (the CPU actually keeps track of the address of instructions to execute them in sequence). Instructions contain the addresses of the variables to manipulate them.

Memory is divided into regions – part of the memory will contain the program to execute (the machine instructions), part of the memory is used for executing the sub-programs of a program (such memory is retrieved after the execution of a sub-program and re-used during the execution of other sub-programs), and part of the memory is used to store variables and other data structures that are available during the whole execution of a program (for example, for storing arrays and objects). The program model represents each of these memory parts as the **program memory**, the **working memory**, and the **global memory**. Note that the working memory is divided into pieces to illustrate how working memory is "reserved" for the execution of different sub-programs.

The following two pages provide blank working models so that you may use them for studying and completing assignments. The first page does not include global memory and can be used during most of the first half of the course since global memory is used when arrays are introduced.


Program Memory
Working memory
CPU


Program Memory
Working memory
Global Memory
CPU

3. Pre-defined Algorithm Models

The following algorithm models can be used for developing algorithms. Most of the algorithms represent input and output for interacting with the user. The last three algorithms represent other tasks supported by the computer (or language).

PrintLine(<argument list>)
: Prints on the terminal screen the contents of the argument list. The argument list consists of a comma separated list of items that can be a string (e.g. " a string") or a variable name. After the output, the cursor on the terminal is placed on the next line.

Print(<argument list>)
: Prints on the terminal screen the contents of the argument list. The argument list consists of a comma separated list of items that can be a string (e.g. " a string") or a variable name. After the output, the cursor remains after the last character printed.

NumVar ← ReadReal()
: Reads a real number from the keyboard

IntVar ← ReadInteger()
: Reads an integer number from the keyboard

StrVar ← ReadLine()
: Reads a line of input, i.e. a string from the keyboard

BoolVar ← ReadBoolean()
: Reads "true" or "false" and stores TRUE or FALSE; it gives a Boolean value as a result.

CharVar ← ReadCharacter()
: Reads a single character from the keyboard.

NumVar ← Random():
: Generates a random value (result NumVar) greater or equal to 0.0 and less than 1.0.

anArrayRefVar ← MakeNewArray(L)
: Creates an array, referenced by anArrayRefVar, which has L positions with unknown values in them.

aMatrixRefVar ← MakeNewMatrix(R, C)
: Creates a matrix, referenced by aMatrixRefVar, which has R rows and C columns of elements with unknown values in them.

4. Summary of Program Structure, Algorithm Concepts and Java Programming Concepts

Algorithm Model

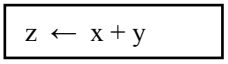
Instruction Block

Subprogram

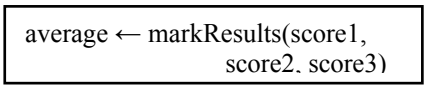
…
Result ← Name(<*paramList*>)

Instructions

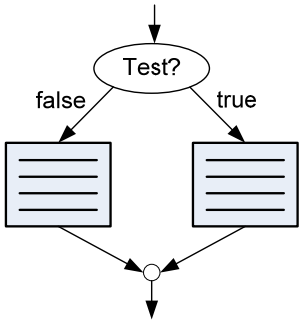
Simple Expression



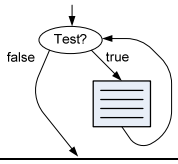
Calls to subprograms



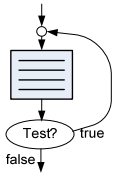
Branch Structure



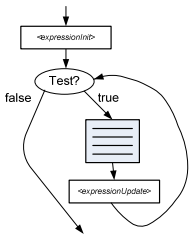
Pre-Test Loop Structure



Post-Test Loop Structure



Alternative Loop Structure (Java for loop)



Java

Instruction Block

```
{
   <instruction>
   <instruction>
        .
        .
}
```

Subprogram

```
public static <type>
name(<paramList>)
{
   <instruction>
   <instruction>
        .
        .
}
```

Instructions

Simple Expression

```
z = x + y;
```

Calls to subprograms

```
average = markResults(score1,
             score2, score3);
```

Branch Structure

```
if (<logical expression>)
{
   <instruction>
   <instruction>
        .
        .
}
else
{
   <instruction>
   <instruction>
        .
        .
}
```

Pre-Test Loop Structure

```
while(<test>)
{
   <instruction>
   <instruction>
        .
        .
}
```

Post-Test Loop Structure

```
do
{
   <instruction>
   <instruction>
        .
        .
}
while(<test>);
```

Alternative Loop Structure (Java for loop)

```
for(<expressionInit>;
    <test>;
    <expressionUpdate>)
{
   <instruction>
   <instruction>
        .
        .
}
```

Arrays / Matrices

```
// Array
Type [] varname;
varname = new Type[size];

// Matrix
Type [][] varname;
varname = new Type[#Rows][#Cols];

// Type – any variable type (int,
// double, etc.) or class name
// varname – legal variable name
// size – number of elements
// #Rows – number of rows
// #Cols – number of columns
```

Classes / objects

```
class ClassName
{
   // Attributes

   // Constructor
   public ClassName(parameterList)
   {
   }

   // Methods
   .
   .
   .
}
// ClassName – valid name
// keywords used in
// declaring attributes
// and variables
public – accessible by all
private – accessible by class only
static – class instance or method
// Reference variable
ClassName varname;
// Creating an object
varname = new ClassName();
```

Available Class/Methods

ITI1120 Methods:

```
ITI1120.readInt( )
ITI1120.readDouble( )
ITI1120.readChar( )
ITI1120.readBoolean( )
ITI1120.readDoubleLine( )
ITI1120.readIntLine( )
ITI1120.readCharLine( )
ITI1120.readString( )
```

Scanner Methods:

```
Scanner keyboard =
          new Scanner(System.in);
keyboard.nextInt( )
keyboard.nextDouble( )
keyboard.nextBoolean( )
keyboard.nextLine( )
```

System.out Methods:

```
System.out.println( … )
System.out.print( … )
```

Math Class Methods/Attributes

```
Math.sin(rad) //cos, tan, asin,
              // acos, PI
Math.exp(x) // also log(x)
Math.log(x) // also log10(x)
Math.pow(x, a) // x to power a
Math.sqrt(x)
Math.random()
Math.abs(x)
```

5. Section 1 Exercises

Program Memory

Exercise 1-1 - Algorithm for Average

Working memory

GIVENS: Num1, Num2, Num3 (three numbers)
RESULTS:
Avg (the average of Num1, Num2, and Num3)
HEADER:

BODY:

CPU

Program Memory **Exercise 1-2 - Another example** **Working memory**

GIVENS:

RESULTS:

HEADER:

BODY:

GIVENS:

INTERMEDIATES:

RESULTS:

HEADER:

BODY:

CPU

Program Memory

Exercise 1-3 Average out of 100

Working memory

Givens:

Results:

Intermediates:

Header:

Body:

CPU

Program Memory

Exercise 1-4 Last Example

Working memory

… Without a constant

…

Body

…

T1 ← C1 * 0.07 // GST

T2 ← C2 * 0.07 // GST

T3 ← C3 * 0.07 // PST

T4 ← C4 * 0.07 // GST

…

…with constants GST and PST

CPU

6. Section 2 Exercises

Exercise 2-1 – Collating Sequence

`'A' < 'a'` is

while

`'?' < ' '` is

Exercise 2-2 - Test for Upper case

- Suppose the variable **x** contains a value of type **char**.
- Write a Boolean expression that is TRUE if the value of **x** is an upper case letter and is FALSE otherwise.
  - Note that you don't need to know the actual code value of the characters!

Exercise 2-3 - Operator Precedence

- What is the order of evaluation in the following expressions?

```
a + b + c + d + e
```

```
a + b * c - d / e
```

```
a / (b + c) - d % e
```

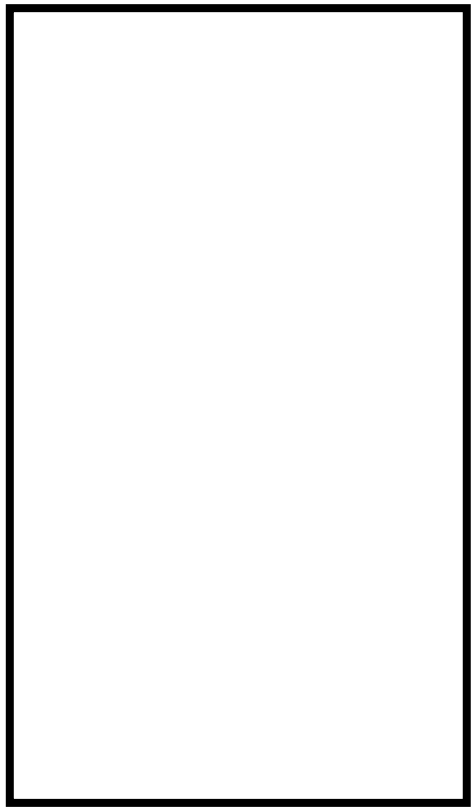
```
a / (b * (c + (d - e)))
```

Program Memory

Exercise 2-4 - Some math

Working memory

```
// Variable declarations



// Compute the average

// Variable declarations


//Compute the quotient and remainder
```



CPU

7. Section 3 Exercises

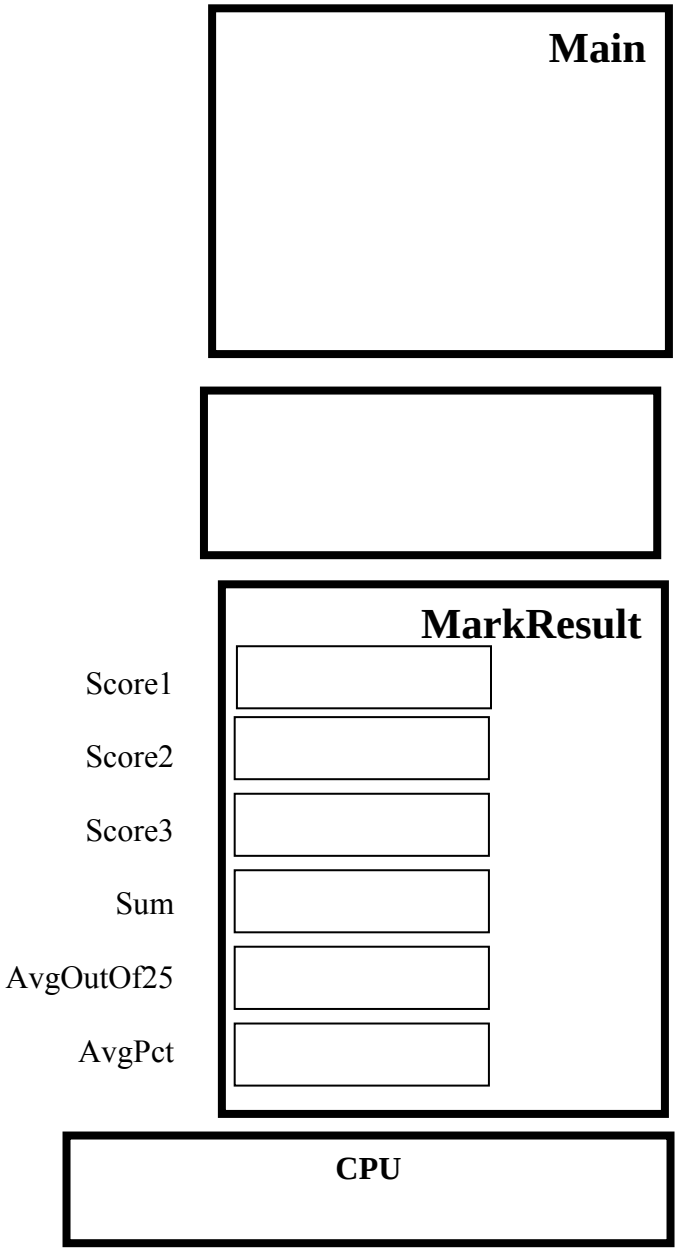
Program Memory **Exercise 3-1 – Main Algorithm** **Working memory**

Givens:
Results:
Intermediates:

Header: M**ain()**
Body:

Givens: **Score1, Score2, Score3** (scores out of 25)
Results: **AvgPct** (average of scores, out of 100)
Intermediates: **Sum** (sum of scores)
AvgOutOf25 (average of scores, out of 25)
Header: **AvgPct ← MarkResult(Score1, Score2, Score3)**
Body:

1. **Sum ← Score1 + Score2 + Score3**
2. **AvgOutOf25 ← Sum / 3**
3. **AvgPct ← AvgOutOf25 * 4**



Program Memory

Exercise 3-2 – Swap 2 Values

Working memory

GIVENS:

RESULTS:

MODIFIEDS:

INTERMEDIATES:

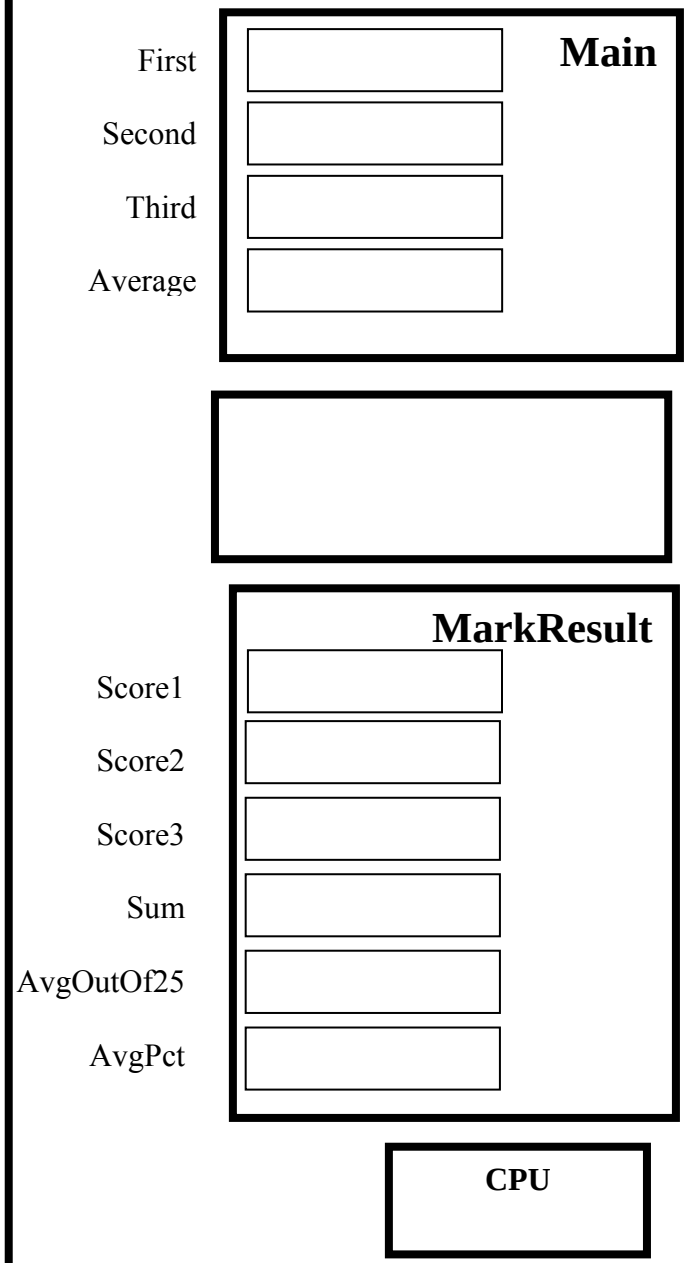
HEADER

BODY:

CPU

Program Memory **Exercise 3-3 – Translating to Java** **Working memory**

```
public static void main (String[] args)
{
   // SET UP KEYBOARD INPUT
   Scanner keyboard = new Scanner( System.in );
   // DECLARE VARIABLES/DATA DICTIONARY


   // READ IN Values from the user




   // Call to markResults

   // PRINT OUT RESULTS

}
public static double markResult(double score1,
                                double score2,
                                double score3 )
{
   // Intermediate variables


   // Result variable

   // BODY OF ALGORITHM



   // RETURN RESULTS

}
```



8. Section 4 Exercises

Program Memory **Exercise 4-1 Tracing Example** **Working memory**

Call: AvgPct ← MarkResult(18, 23, 19)

Givens: Score1, Score2, Score3 (scores out of 25)
Results: AvgPct (average of scores, out of 100)
Intermediates:
Sum (sum of scores)
AvgOutOf25 (average of scores, out of 25)
Header: AvgPct ← MarkResult(Score1, Score2, Score3)
Body:
1. Sum ← Score1 + Score2 + Score3
2. AvgOutOf25 ← Sum / 3
3. AvgPct ← AvgOutOf25 * 4

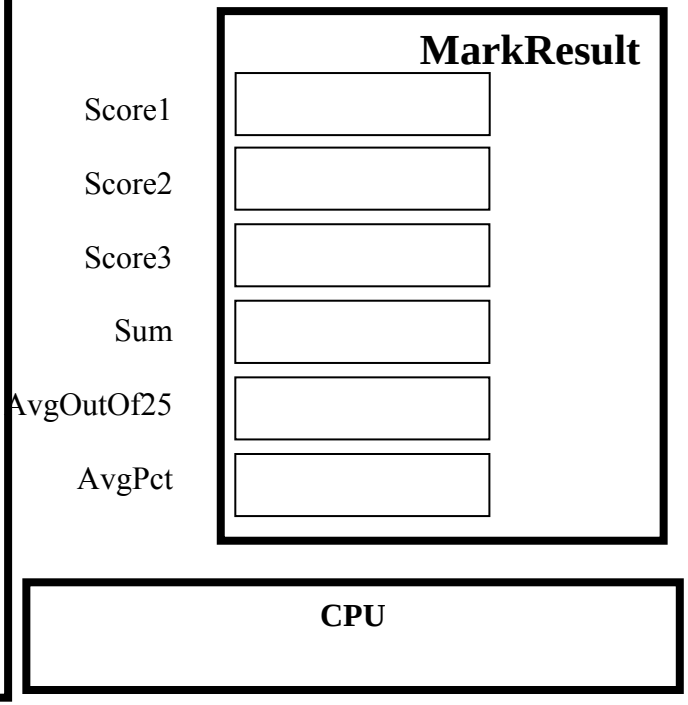


Table 1: Trace for AvgPct ← MarkResult(18, 23, 19)

Statement	Score1	Score2	Score3	Sum	AvgOutOf25	AvgPct
Initial values						
1. Sum ← Score1 + Score2 + Score3						
2. AvgOutOf25 ← Sum / 3						
3. AvgPct ← AvgOutOf25 * 4						

Program Memory **Exercise 4-2 Tracing a Call** **Working memory**

Givens: none
Results: none
Intermediates:
First, Second, Third (three scores)
Average (average of scores, out of 100)
Header: M**ain()**
Body:
(Read in scores from the user)

1. **PrintLine("Please enter three scores")**
2. **First ← ReadReal()**
3. **Second ← ReadReal ()**
4. **Third ← ReadReal ()**
   (Call the MarkUser algorithm)
5. **Average ← MarkResult(First, Second, Third)**
   (Print the average for the user)
6. **PrintLine("The average is ", Average)**

Givens: **Score1, Score2, Score3** (scores out of 25)
Results: **AvgPct** (average of scores, out of 100)
Intermediates: **Sum** (sum of scores)
AvgOutOf25 (average of scores, out of 25)
Header: **AvgPct ← MarkResult(Score1, Score2, Score3)**
Body:

1. **Sum ← Score1 + Score2 + Score3**
2. **AvgOutOf25 ← Sum / 3**
3. **AvgPct ← AvgOutOf25 * 4**

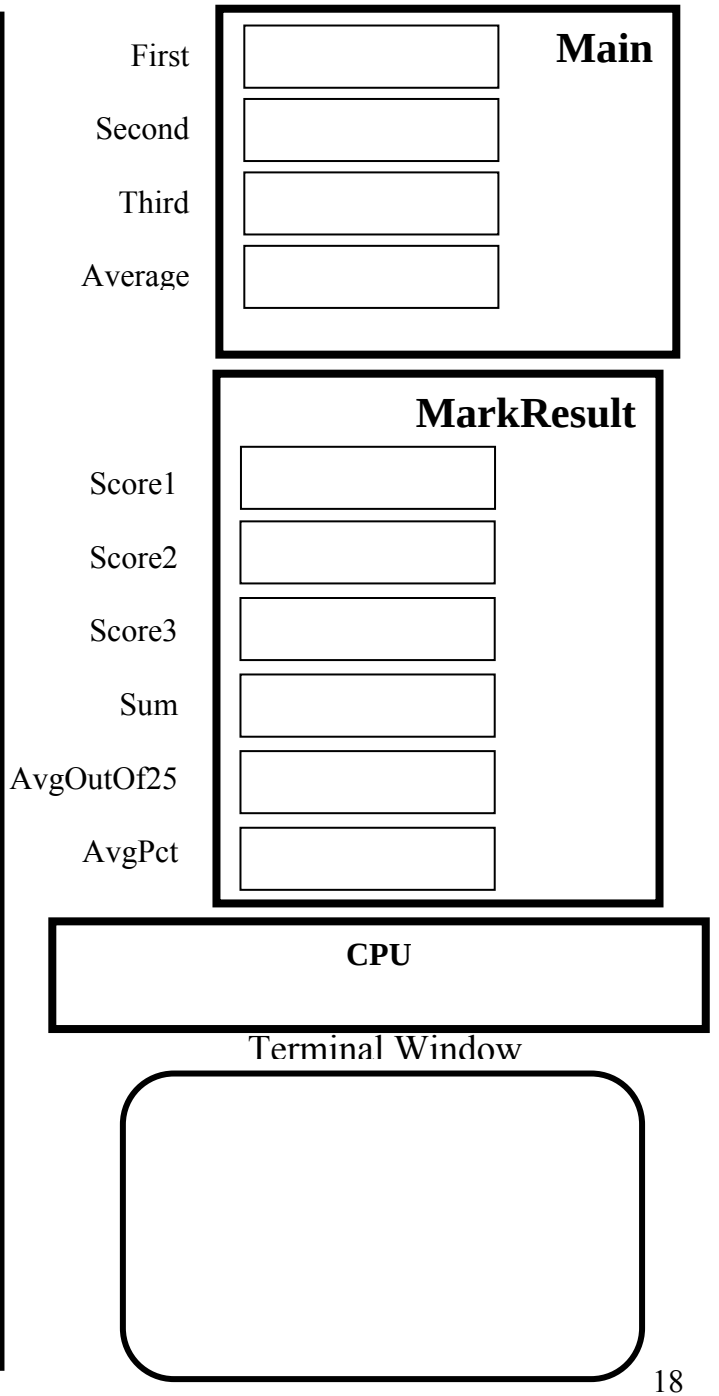


Table 1: Trace for main algorithm:

Interaction with user:

Please enter three scores out of 25

23 16 21

The average is 80 percent

Statements	First	Second	Third	Average
Initial values				
1. PrintLine("Please enter three scores")				
2. First ← ReadReal()				
3. Second ← ReadReal ()				
4. Third ← ReadReal ()				
5. Call Average ← MarkResult(First, Second, Third) (See Table 2)				
6. PrintLine("The average is ", Average)				

Call algorithm MarkResult:

Average ← MarkResult (First, Second, Third)

AvgPct ← MarkResult (Score1, Score2, Score3)

Table 2 – Trace for AvgPct ← MarkResult (23. 16. 21)

Statement	Score1	Score2	Score3	Sum	AvgOutOf25	AvgPct
Initial values						
1. Sum ← Score1 + Score2 + Score3						
2. AvgOutOf25 ← Sum / 3						
3. AvgPct ← AvgOutOf25 * 4						

Program Memory **Exercise 4-3 – Marks out of 100, again** **Working memory**

Givens:

Results:

Intermediates:

Header:

Body:

GIVENS: Num1, Num2, Num3 (three numbers)
RESULTS: Avg (the average of Num1, Num2, and Num3)
HEADER: Avg ← Average(Num1, Num2, Num3)
BODY:

1. Avg ← (Num1 + Num2 + Num3)/3

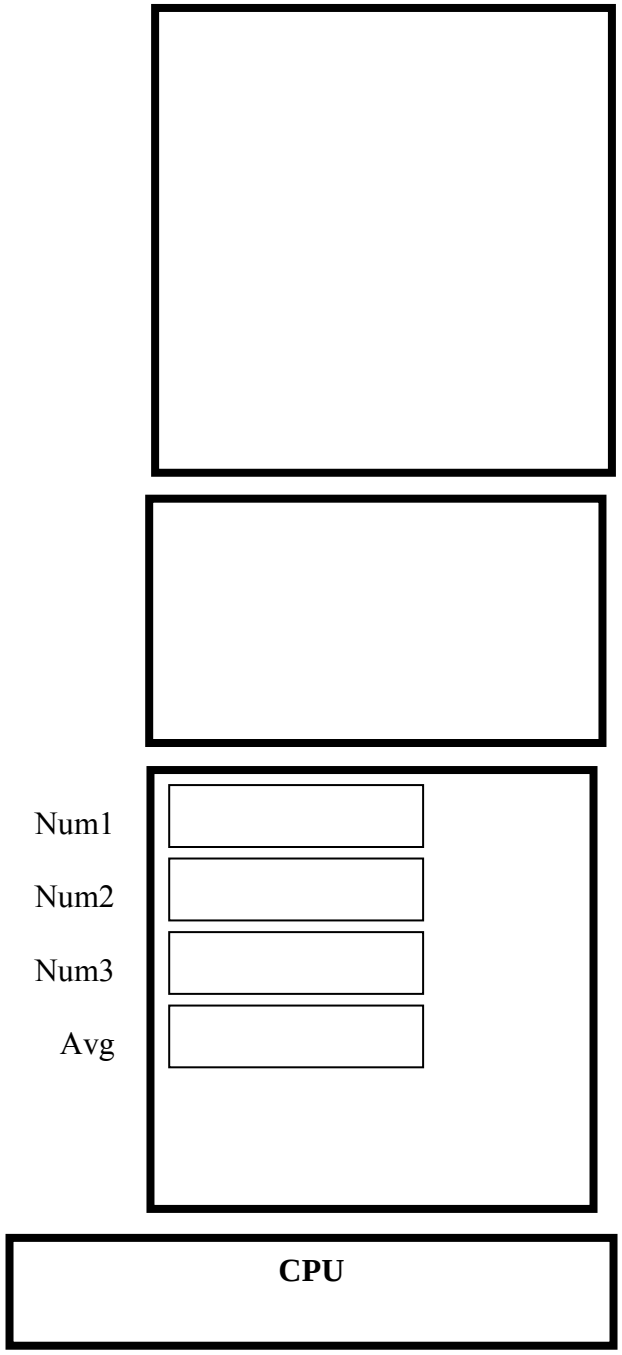


Table 1 - Table for AvgPct ← MarkResult(23, 16, 21)

Statements					
Initial values					

Call algorithm Average:

Table 2 - Table for Avg ← Average( )

Statement	Num1	Num2	Num3	Sum	Avg
Initial values					
1. Sum ← Num1 + Num2 + Num3					
2. Avg ← Sum / 3					

Program Memory

Exercise 4-5 – Reverse Digits

Working memory

GIVENS:

RESULTS:

IINTERMEDIATES:

HEADER:

BODY:

The following algorithm is available to extract the ten's and one's digits from a two digit number:

(High, Low) ← Digits(X)

CPU

Table1 - Trace for ReverseN ← Rev2(42))

Statements	N	Tens	Ones	ReverseN
Initial values				
1. Call Digits(N)				
2. ReverseN ← Ones * 10 + Tens				

Call to (Tens, Ones) ← Digits (N)
(Tens, Ones) ← Digits (N)

(High, Low) ← Digits (X)

Program Memory

Exercise 4-7 – Join Four Numbers

Working memory

Givens:

Result:

Intermediates:

Header:

Body:

You may assume there is available an algorithm:

C ← Join(A, B)

Givens: A, B, two positive integers
Result: C is the number having the digits in A followed by the digits in B.

CPU

9. Section 5 Exercises

Program Memory

Exercise 5-1 - Back to the Larger of Two Numbers

Working memory

GIVENS:

RESULT:

HEADER:

BODY:

CPU

Exercise 5-2 - Maximum of 3 numbers

GIVENS:

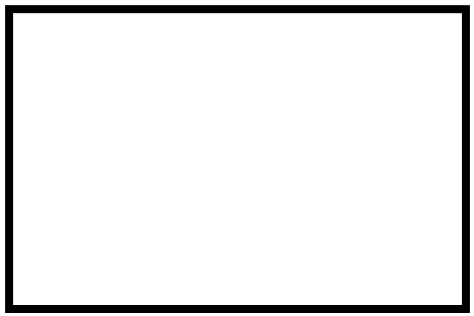
RESULT:

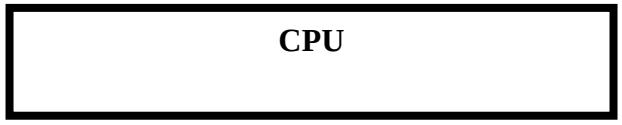
HEADER:

BODY:

(Nested tests)

OR
(Sequence of tests)

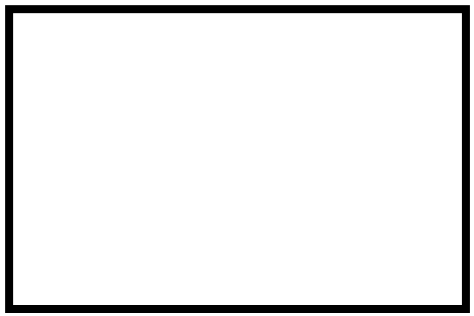




GIVENS: X, Y, Z (three numbers)
RESULT: M (the larger of X, Y and Z)
HEADER: M ← Max3(X, Y, Z)
BODY:

Sequence of tests

```
public double max3(double x,
        double y, double z)
{
    double m;

    return(m);
}
```

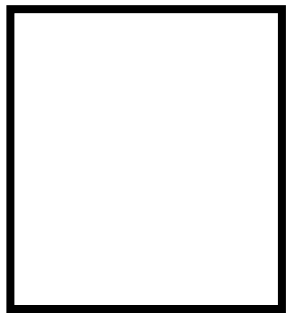


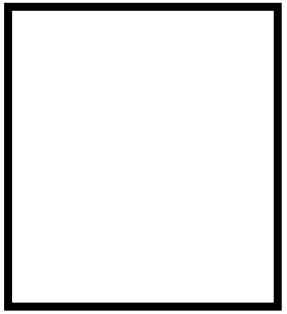
CPU

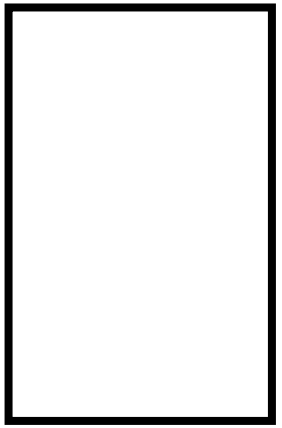
GIVENS: X, Y, Z (three numbers)
RESULT: M (the larger of X, Y and Z)
HEADER: M ← Max3(X, Y, Z)
BODY:

Nested tests

```
public double max3(double x,
        double y, double z)
{
    double m;

    return(m)
}
```







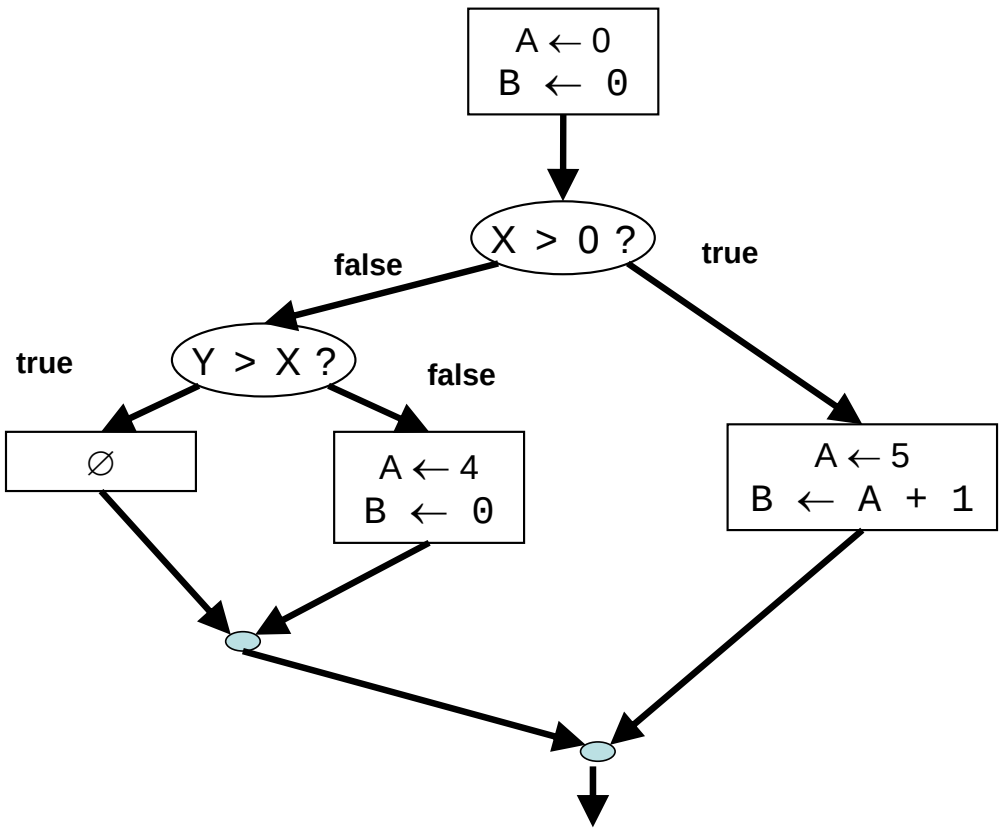
CPU

Program Memory

Exercise 5-5 – Translation of an Instruction Bloc

Working memory

{

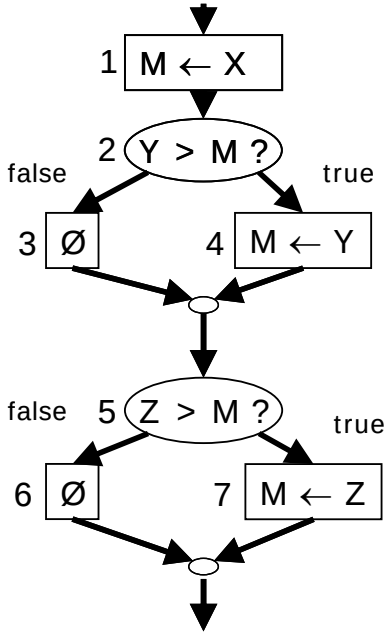


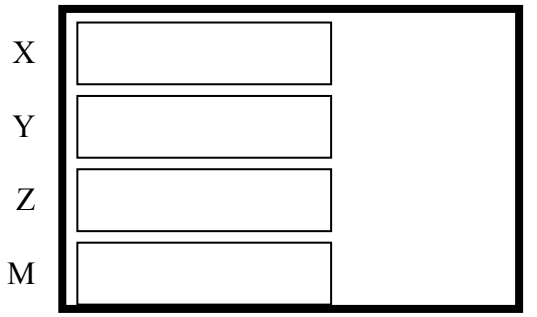
}

CPU

Program Memory **Exercise 5-6 - Trace of MAX3, version 2** **Working memory**

GIVENS: X, Y, Z (three numbers)
RESULT: M (the larger of X, Y and Z)
HEADER: M ← Max3(X, Y, Z)
BODY:





CPU

	X	Y	Z	M
Initial values				

Program Memory

Exercise 5-7 - Movie Tickets

Working memory

GIVENS: Age (persons age)
RESULT: Cost (ticket cost)
HEADER: Cost ← TicketCost(Age)
BODY:

(Version 1: Nested Tests)

(Version 2: Sequence of Tests)

CPU

Program Memory

Exercise 5-8 - Positive Value

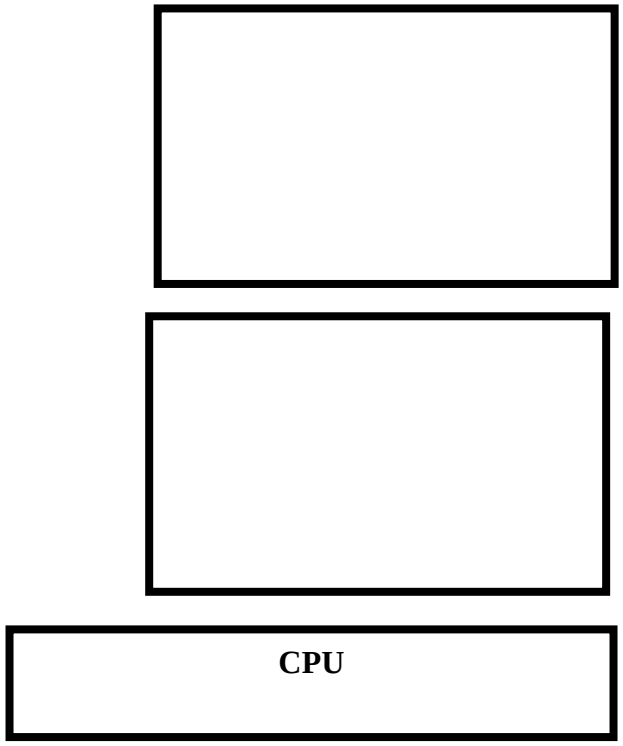
Working memory

GIVEN:

RESULT:

HEADER:

BODY:



Exercise 5-9 Compound Boolean Expressions

Exercise 5-10 - More Compound Boolean Expressions

Suppose X = 5 and Y = 10.

Expression	Value
(X > 0) AND (NOT (Y = 0))	
(X > 0) AND ((X < Y) OR (Y = 0))	
(NOT (X > 0)) OR ((X < Y) AND (Y = 0))	
NOT ((X > 0) OR ((X < Y) AND (Y = 0)))	