

Computation Theory

Solution Notes for Example Sheet 1

In this document you will find some solution notes for the problems of Example Sheet 1 of Computation Theory. If you find any mistake or any typos, please do let me know. Also, I am happy to hear (and include them in the notes (with credit) if you want) about alternative solutions to the problems or variations of a problem that you came up with.

Lecture 1

Exercise 1 What are decision problems?

A decision problem has the form: Given an element $s \in S$ and a property P , output 0 or 1 depending on whether $P(s)$ holds.

See Lecture 1 slide 9.

Exercise 2 Define the *Halting problem*. Explain the informal argument of why the Halting problem is undecidable.

The halting problem is the decision problem concerning the set of all (program, input) pairs and the property is whether the program when given the specific input terminates.

See Lecture 1 slide 15.

Assume that there exists a program H that can decide this property. Then, consider the program H' that modifies H so that if $H(x, x)$ outputs 1, then H' loops (otherwise it halts). Consider the pair $v = (H', H')$. Then $H((H', H'))$ terminates (since H terminates for all inputs), so $H'(H')$ terminates, so $H((H', H'))$ does not terminate (contradiction).

See Lecture 1 slide 19.

What are the assumptions that we are making? Why is this not rigorous?

Lecture 2

Exercise 3

- Define a *register machine (RM)*. (See [2010P6Q3 (a)] or [2007P3Q7 (a)(i)])
- Define a *register machine configuration*. (See [2009P6Q3 (a)] or [1999P3Q9 (b)])
- Define a *register machine computation* (See [2010P6Q3 (a)] or [1999P3Q9 (a)]). What do we mean when we say that the execution of an RM is *deterministic*?
- What are the two ways that an RM *halts*?
- How can you modify a program to turn all erroneous halts into proper halts?

- A register machine M consists of finitely many *registers* R_0, \dots, R_m each capable of storing a natural number, together with a program P consisting of a finite list of *instructions*

$$L_0 : body_0, L_1 : body_1, \dots, L_n : body_n$$

where each $body_i$ takes one of three possible forms:

- $R_i^+ \rightarrow L$: add 1 to the contents of the i th register and then jump to the instruction labelled L ;
- $R_i^- \rightarrow L, L'$: if the contents of the i th register is > 0 , then subtract 1 from it and jump to the instruction labelled L , otherwise jump to the instruction labelled L' ;
- HALT*: stop executing instructions.

See Lecture 2 slide 5.

(b) A *configuration* $(\ell, [r_0, r_1, \dots, r_n])$ of a register machine consists of the label ℓ of the current program instruction together with a list $[r_0, r_1, \dots, r_n]$ of the current contents of the machine's registers R_0, R_1, \dots, R_n .

See **Lecture 2 slide 17.**

(c) A computation of a RM is a (finite or infinite) sequence of configurations c_0, c_1, c_2, \dots where

- $c_0 = (0, r_0, \dots, r_n)$ is an initial configuration
- each $c = (\ell, r_0, \dots, r_n)$ in the sequence determines the next configuration in the sequence (if any) by carrying out the program instruction labelled L_ℓ with registers containing r_0, \dots, r_n .

By deterministic, we mean that for a given program, a given configuration and a given instruction ℓ , there is a unique next configuration.

See **Lecture 2 slide 18.**

(d) A RM halts when it reaches a halting configuration, i.e.

- either ℓ^{th} instruction in program has body HALT (a “proper halt”)
- or ℓ is greater than the number of instructions in program, so that there is no instruction labelled L_ℓ (an “erroneous halt”)

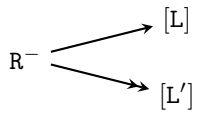
See **Lecture 2 slide 21.**

(e) We compute the number L of instructions in the program and then loop through the branch instructions, checking if the target labels are $< L$. Those that are $\geq L$, we replace with $L + 1$ and insert a HALT instruction at the end of the program. This means that erroneous halts will be converted to proper HALTS at instruction L .

(Alternative- a bit more wasteful), append HALT instructions to the program until all labels are valid.

Exercise 4 [RM graphical representation] Define the graphical representation for the RM programs.

In the graphical representation for the RM programs, nodes represent instructions and arcs represent jumps between instructions. Let $[L]$ denote the node of the instruction at label L , then the instructions are represented as follows:

instruction	representation
$R^+ \rightarrow L$	$R^+ \longrightarrow [L]$
$R^- \rightarrow L, L'$	
HALT	HALT
L_0	START $\longrightarrow [L_0]$

See **Lecture 2 slide 23.**

Given a graphical representation, how can you recover the RM program?

Exercise 5

- (a) Define a *partial function*.
- (b) Why is the relation between initial and final register contents of a RM a partial function?
- (c) Define a *total function*.
- (d) Show that a total function is always a partial function.
- (e) Define the notations $f(x) \downarrow$, $f(x) \uparrow$, $X \rightarrow Y$ and $X \dashrightarrow Y$.
- (f) Give an example of an RM program that is a total function and an RM program that is partial (but not total).

(a) A partial function from X to Y is any subset $f \subseteq X \times Y$ such that

$$\forall x \in X. \forall y \in Y. (x, y) \in f \wedge (x, y') \in f \Rightarrow y = y'.$$

See Lecture 2 slide 30.

(b) The RM computations are deterministic so, if for some register values R there exists a sequence to a halt, then the configuration will be unique, so the condition of a partial function is satisfied.

(c) A partial function $f : X \rightarrow Y$ is total iff $\forall x \in X. \exists y \in Y. (x, y) \in f$ (or equivalently $\forall x \in X. f(x) \downarrow$).

See Lecture 2 slide 33.

(d) A total function is a partial function (by definition) that satisfies the additional property that every x has a mapping.

- (e)
- $f(x) \downarrow \Leftrightarrow \exists y \in Y. f(x) = y$,
 - $f(x) \downarrow \Leftrightarrow \neg \exists y \in Y. f(x) = y$,
 - $X \rightarrow Y := \{f \mid f \text{ a total function}\}$,
 - $X \rightarrow Y := \{f \mid f \text{ a partial function}\}$.

See Lecture 2 slide 32.

(f) A trivial example of a total function $f(x) = 0$, represented by

START \longrightarrow HALT

and a trivial example of a partial function (that is not total) is the completely undefined function:

START \longrightarrow R_0^+

Exercise 6 [RM Computable]

- (a) What does it mean for a function to be *RM computable*? (See [2013P6Q3 (b)] or [2010P6Q3 (b)] or [2007P3Q7 (b)(i)] or [2005P3Q7 (c)])
- (b) What is the *IO convention*?

(a) $f(x_1, \dots, x_n)$ is register machine computable if there is a register machine M with at least $n + 1$ registers such that for all $(x_1, \dots, x_n) \in \mathbb{N}^n$ and all $y \in \mathbb{N}$, the computation of M starting with $R_0 = 0, R_1 = x_1, \dots, R_n = x_n$ and all other registers zeroed halts with $R_0 = y$ if and only if f is defined at (x_1, \dots, x_n) and takes value y there.

See Lecture 2 slide 34.

(b) The *IO convention* states that registers R_1, \dots, R_n in the initial configuration store the function's arguments (with all others zeroed); and in the halting configuration register R_0 stores its value (if any).

See Lecture 2 slide 34.

Exercise 7

In this exercise, you will investigate a simple register machine emulator.

- (a) In the “rm.zip” file execute the “Examples.java” file and see the execution for an add and a copy program.
- (b) Write a similar program for multiplying two integers and execute it. (Use the `debug` flag to output the configuration sequence and the flag `maxIter` to set an upper bound on the computation steps).

```
public static void multiplySolution() {
    Program program = new Program(Arrays.asList(
        SUB(R(1), L(1), L(6)),
        SUB(R(2), L(2), L(4)),
        ADD(R(0), L(3)),
        ADD(R(3), L(1)),
    ));
}
```

```

        SUB(R(3), L(5), L(0)),
        ADD(R(2), L(4)),
        HALT
    ), doNotAllowErroneousHalts);
RegisterMachine rm = new RegisterMachine();
rm.setDebug(true);
RegisterConfiguration config = new RegisterConfiguration(4);
config.setRegister(1, BigInteger.valueOf(2));
config.setRegister(2, BigInteger.valueOf(3));
rm.execute(program, config);
System.out.println(config);
}

```

The debug output for inputs 2 and 3 was:

```

(0, [R0=0;R1=2;R2=3;R3=0;])
(1, [R0=0;R1=1;R2=3;R3=0;])
(2, [R0=0;R1=1;R2=2;R3=0;])
(3, [R0=1;R1=1;R2=2;R3=0;])
(1, [R0=1;R1=1;R2=2;R3=1;])
(2, [R0=1;R1=1;R2=1;R3=1;])
(3, [R0=2;R1=1;R2=1;R3=1;])
(1, [R0=2;R1=1;R2=1;R3=2;])
(2, [R0=2;R1=1;R2=0;R3=2;])
(3, [R0=3;R1=1;R2=0;R3=2;])
(1, [R0=3;R1=1;R2=0;R3=3;])
(4, [R0=3;R1=1;R2=0;R3=3;])
(5, [R0=3;R1=1;R2=0;R3=2;])
(4, [R0=3;R1=1;R2=1;R3=2;])
(5, [R0=3;R1=1;R2=1;R3=1;])
(4, [R0=3;R1=1;R2=2;R3=1;])
(5, [R0=3;R1=1;R2=2;R3=0;])
(4, [R0=3;R1=1;R2=3;R3=0;])
(0, [R0=3;R1=1;R2=3;R3=0;])
(1, [R0=3;R1=0;R2=3;R3=0;])
(2, [R0=3;R1=0;R2=2;R3=0;])
(3, [R0=4;R1=0;R2=2;R3=0;])
(1, [R0=4;R1=0;R2=2;R3=1;])
(2, [R0=4;R1=0;R2=1;R3=1;])
(3, [R0=5;R1=0;R2=1;R3=1;])
(1, [R0=5;R1=0;R2=1;R3=2;])
(2, [R0=5;R1=0;R2=0;R3=2;])
(3, [R0=6;R1=0;R2=0;R3=2;])
(1, [R0=6;R1=0;R2=0;R3=3;])
(4, [R0=6;R1=0;R2=0;R3=3;])
(5, [R0=6;R1=0;R2=0;R3=2;])
(4, [R0=6;R1=0;R2=1;R3=2;])
(5, [R0=6;R1=0;R2=1;R3=1;])
(4, [R0=6;R1=0;R2=2;R3=1;])
(5, [R0=6;R1=0;R2=2;R3=0;])
(4, [R0=6;R1=0;R2=3;R3=0;])
(0, [R0=6;R1=0;R2=3;R3=0;])
(6, [R0=6;R1=0;R2=3;R3=0;])
[R0=6;R1=0;R2=3;R3=0;]

```

Exercise 8 (optional) In this exercise, you can experiment and implement various extensions to the RM emulator (of course you can create your own from scratch).

- (a) Add an instruction that adds two register values and stores the result in the third one. (This is mostly to understand how the current emulator works, you will not need this for subsequent steps)

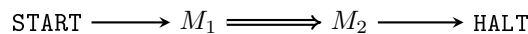
- (b) Create a function that takes a program and a mapping for the registers (e.g. $\{R_1 \rightarrow R_2, R_2 \rightarrow R_4, R_5 \rightarrow R_3\}$) and returns a new program where operations are performed on the mapped registers.
- (c) Write a function that concatenates various programs together, so that one executes after the other. Note: You need to change the labels and also replace HALT instructions.
- (d) Use the previous two operations to create a program that uses copy (defining it once) and add.
- (e) Write a function that takes three programs M_1 , M_2 and M_3 and returns a program that implements if M_1 then M_2 else M_3 . (You need to define its semantics)
- (f) Write a function that takes two programs M_1 and M_2 and returns a program that implements while M_1 do M_2 .
- (g) Write a function that takes a program M and three registers and implements for $R_i = R_1$ to R_2 do M .
- (h) Write a program that computes the Fibonacci numbers using the high-level constructs.
- (i) (optional - more of Compiler Design) Implement a parser for reading RM programs and converting them to abstract syntax trees (which are just lists in this case).

Exercise 9 [High-level constructs] Show that the following high-level constructs can be implemented using RMs.

- (a) (sequential composition) If M_1 and M_2 are programs, then there exists a program M_3 that is equivalent to first executing M_1 and then M_2 .
- (b) (if-then-else statements) If M_1 , M_2 and M_3 are programs, then there exists a program M_4 that is equivalent to **if** M_1 **then** M_2 **else** M_3 . (You need to define the exact semantics for if)
- (c) (while-do) If M_1 and M_2 are programs, then there exists a program M_3 that is equivalent to **while** M_1 **do** M_2 . (Again, you need to define the exact semantics for while)
- (d) (optional) Similarly, define a program that is equivalent to a for-loop.
- (e) Give a high-level argument for why any function computable using a high-level programming language is computable using a RM. Is the other direction true?

Before we begin, let us introduce the \implies notation in RM diagrams, where $M \implies M'$ means replacing all HALT and redirecting all erroneous halts of M to the first instruction of M' . (The redirection can be implemented by adding to an irrelevant register and then subtracting from that)

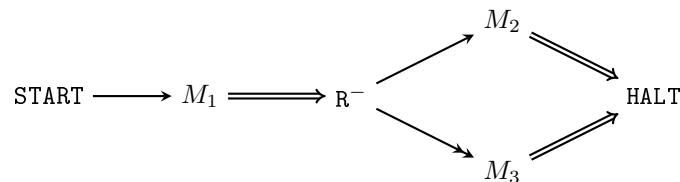
- (a) The composition of two RMs M_1 and M_2 is simply going to be:



If we want to obtain a program from this, then we also need to add an offset to all labels of M_1 .

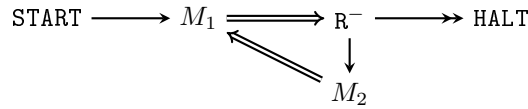
See Lecture 2 slide 43.

- (b) Assume that M_1 is a program that sets register R to either 0 or 1 indicating whether its function is false or true respectively. Then we define the semantics of the **if**-statement to be: execute M_1 if $R = 1$ otherwise execute M_2 if $R = 0$. This is achieved by the program below:



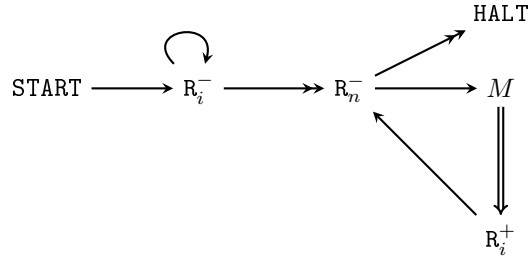
See Lecture 2 slide 43.

- (c) Assume that M is a program that sets register R to either 0 or 1 based on whether its function is false or true. then we define **while** to mean execute M_2 as long as R is 1. This is achieved by the program below:

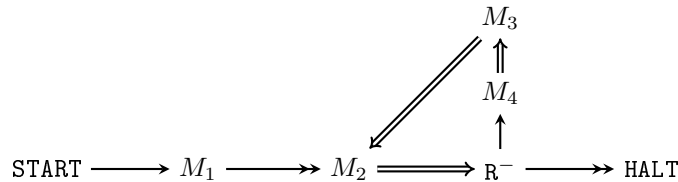


See **Lecture 2 slide 43**.

- (d) There are a few ways to define the for-loop. One is for $R_i = 0$ to n where n cannot be modified, and a program M (which uses R_i is executed). This is achieved by the program below:



An alternative is to implement **for**($M_1; M_2; M_3$){ M_4 } where M_2 is assumed to produce a boolean in register R as above.



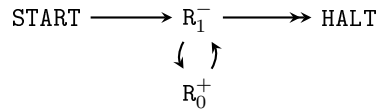
- (e) We can map the control structures to RM programs and using these we can create memories and arrays, so we should be able to implement any (famous) high-level language. The other direction is a bit more subtle. It depends on specifics of the programming language. For example, there should not be any constrain on the size of arrays/memory allocated.

What are some potentials problems that could break these abstractions?

Exercise 10 [Projection] Show that the *first projection* function $p_1^2 : \mathbb{N}^2 \rightarrow \mathbb{N}$, where $p_1^2(x, y) \triangleq x$, is RM computable. Write the RM program and create a diagram for this. Similarly, argue that the *second projection* function $p_2^2(x, y) \triangleq y$ is RM computable.

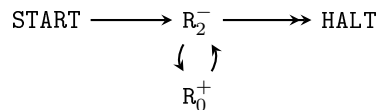
For the first projection function, we have the following code:

$L_0 : R_1^- \rightarrow L_1, L_2$
 $L_1 : R_0^+ \rightarrow L_0$
 $L_2 : \text{HALT}$

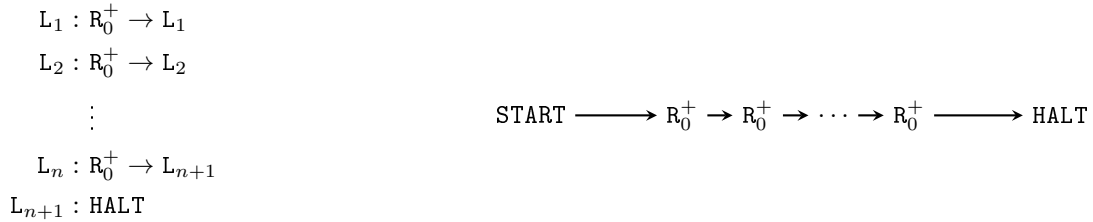


For the second projection function, we change $1 \rightarrow 2$:

$L_0 : R_2^- \rightarrow L_1, L_2$
 $L_1 : R_0^+ \rightarrow L_0$
 $L_2 : \text{HALT}$

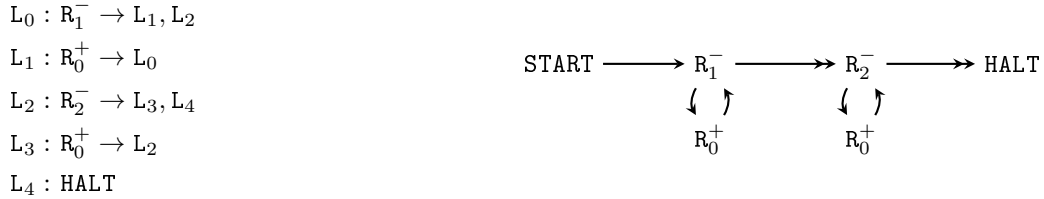


Exercise 11 [Constant] Show that the *constant* function $c : \mathbb{N} \rightarrow \mathbb{N}$, where $c(x) \triangleq n$ for fixed $n \in \mathbb{N}$ is RM computable. Write the RM program and create a diagram for this.



How can you do this with fewer register operations?

Exercise 12 [Addition] Show that the *addition* function $\text{add} : \mathbb{N}^2 \rightarrow \mathbb{N}$, where $\text{add}(x, y) \triangleq x + y$ is RM computable. Write the RM program and create a diagram for this.



See **Lecture 2 slide 36.**

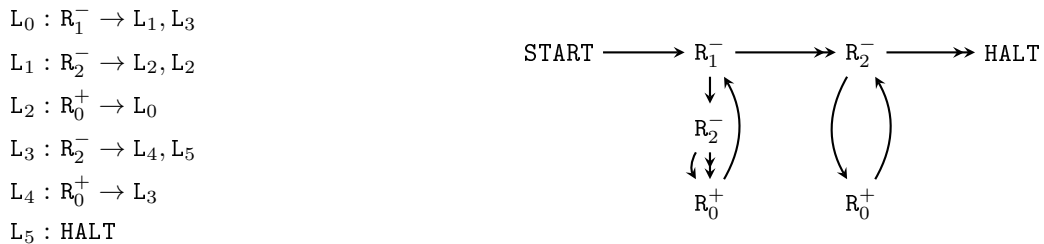
Exercise 13 [Multiplication] Show that the *multiplication* function $\text{mult} : \mathbb{N}^2 \rightarrow \mathbb{N}$, where $\text{mult}(x, y) \triangleq x \cdot y$ is RM computable. Write the RM program and create a diagram for this. (See **[2007P3Q7 (b)(ii)]** for variant)



See **Lecture 2 slide 37.**

Exercise 14 [Max] Show that the *max* function $\text{max} : \mathbb{N}^2 \rightarrow \mathbb{N}$ is RM computable. Write the RM program and create a diagram for this. (See **[2018P6Q6 (b)(ii)]**)

The maximum is R_1 plus the difference $R_2 - R_1$ (if this is positive and 0 otherwise).

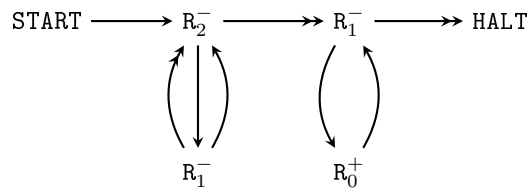


Exercise 15 [Truncated subtraction] Show that the *truncated subtraction* function $\text{tsub} : \mathbb{N}^2 \rightarrow \mathbb{N}$, where

$$\text{tsub}(x, y) \triangleq \begin{cases} x - y & \text{if } y \leq x \\ 0 & \text{if } y > x \end{cases}$$

is RM computable. Write the RM program and create a diagram for this. (See [2010P6Q3 (e)] for variant)

$L_0 : R_2^- \rightarrow L_1, L_2$
 $L_1 : R_1^- \rightarrow L_0, L_0$
 $L_2 : R_1^- \rightarrow L_3, L_4$
 $L_3 : R_0^+ \rightarrow L_2$
 $L_4 : \text{HALT}$



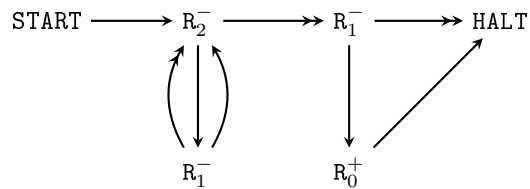
Exercise 16 [Comp] Show that the *comp* function $\text{comp} : \mathbb{N}^2 \rightarrow \mathbb{N}$, where

$$\text{comp}(x, y) \triangleq \begin{cases} 0 & \text{if } x \leq y \\ 1 & \text{otherwise} \end{cases}$$

is RM computable. Write the RM program and create a diagram for this. (See [2018P6Q6 (b)(iii)])

The idea is to compute tsub and then check if it is positive (and output 1) otherwise halt.

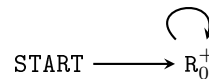
$L_0 : R_2^- \rightarrow L_1, L_2$
 $L_1 : R_1^- \rightarrow L_0, L_0$
 $L_2 : R_1^- \rightarrow L_3, L_4$
 $L_3 : R_0^+ \rightarrow L_4$
 $L_4 : \text{HALT}$



Exercise 17 [Undef] Show that the *undef* function $f : \mathbb{N} \rightarrow \mathbb{N}$ is RM computable. (See [2013P6Q3 (c) (i)])

Any RM that loops forever is good here:

$L_0 : R_0^+ \rightarrow L_0$

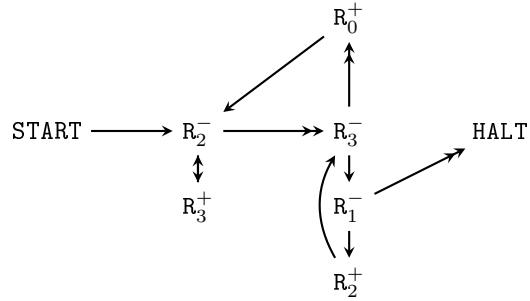


Exercise 18 [Integer division] Show that the *integer division* function $\text{div} : \mathbb{N}^2 \rightarrow \mathbb{N}$, where

$$\text{div}(x, y) \triangleq \begin{cases} \text{quo}(x, y) & \text{if } y > 0 \\ 0 & \text{if } y = 0 \end{cases}$$

is RM computable. Write the RM program and create a diagram for this.

- $L_0 : R_2^- \rightarrow L_1, L_2$
- $L_1 : R_3^+ \rightarrow L_0$
- $L_2 : R_3^- \rightarrow L_3, L_5$
- $L_3 : R_1^- \rightarrow L_4, L_6$
- $L_4 : R_2^+ \rightarrow L_2$
- $L_5 : R_0^+ \rightarrow L_0$
- $L_6 : \text{HALT}$



Exercise 19 [Mod] Show that the *mod* (basically $[\cdot]$, as defined in Discrete Maths) function $\text{mod} : \mathbb{N}^2 \rightarrow \mathbb{N}$, where

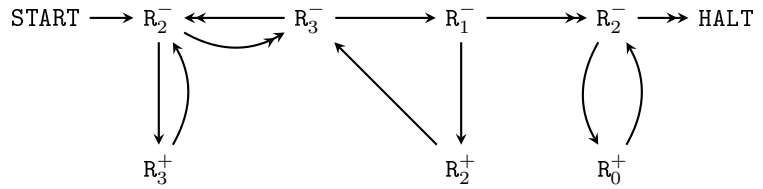
$$\text{mod}(x, y) \triangleq \text{tsub}(x, y \cdot (\text{div}(x, y)))$$

is RM computable. Outline the construction of an RM program that computes this function.

One way is to use the functions that we have defined in previous exercises and combine them as stated in the exercise statement.

Alternatively, we can modify the above code for *div*:

- $L_0 : R_2^- \rightarrow L_1, L_2$
- $L_1 : R_3^+ \rightarrow L_0$
- $L_2 : R_3^- \rightarrow L_3, L_0$
- $L_3 : R_1^- \rightarrow L_4, L_5$
- $L_4 : R_2^+ \rightarrow L_2$
- $L_5 : R_2^- \rightarrow L_6, L_7$
- $L_6 : R_0^+ \rightarrow L_5$
- $L_7 : \text{HALT}$

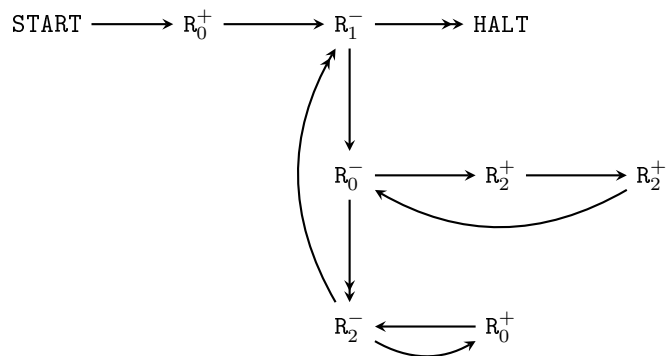


Exercise 20 [Binary Exponential] Show that the *binary exponential* function $e : \mathbb{N} \rightarrow \mathbb{N}$, where

$$e(x) \triangleq 2^x$$

is RM computable. Write the RM program and create a diagram for this. (See [2013P6Q3 (c) (iii)] for variant)

- $L_0 : R_0^+ \rightarrow L_1$
- $L_1 : R_1^- \rightarrow L_2, L_7$
- $L_2 : R_0^- \rightarrow L_3, L_5$
- $L_3 : R_2^+ \rightarrow L_4$
- $L_4 : R_2^+ \rightarrow L_2$
- $L_5 : R_2^- \rightarrow L_6, L_1$
- $L_6 : R_0^+ \rightarrow L_5$
- $L_7 : \text{HALT}$

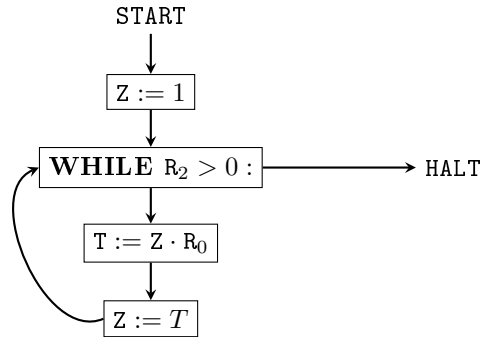


Exercise 21 [Exponentiation] Show that the *exponentiation* function $\text{pow} : \mathbb{N}^2 \rightarrow \mathbb{N}$, where

$$\text{pow}(x, y) \triangleq x^y$$

is RM computable. Outline the construction of an RM program that computes this function.

We can construct an RM using the high-level constructs and map them to an RM program (as discussed in Exercise 9).



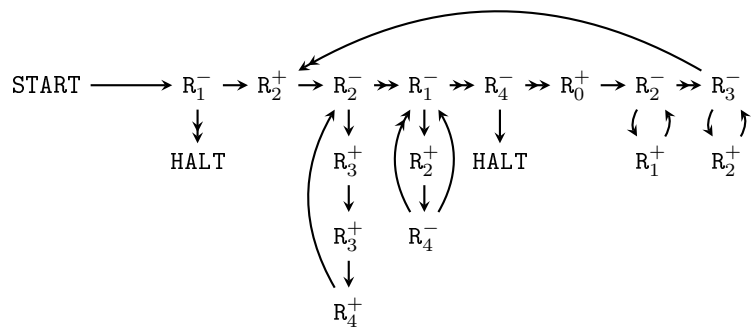
Exercise 22 [Binary Logarithm] Show that the *binary logarithm* function $\log_2 : \mathbb{N} \rightarrow \mathbb{N}$, where

$$\log_2(x) \triangleq \begin{cases} \text{largest } y \text{ such that } 2^y \leq x & \text{if } x > 0 \\ 0 & \text{if } x = 0 \end{cases}$$

is RM computable. Write the RM program and create a diagram for this.

The main idea is to construct powers of two and check if these are greater than the number.

- L₀ : R₁⁻ → L₂, L₁
- L₁ : HALT
- L₂ : R₂⁺ → L₃
- L₃ : R₂⁻ → L₄, L₇
- L₄ : R₃⁺ → L₅
- L₅ : R₃⁺ → L₆
- L₆ : R₄⁺ → L₃
- L₇ : R₁⁻ → L₈, L₁₀
- L₈ : R₂⁺ → L₉
- L₉ : R₄⁻ → L₇, L₇
- L₁₀ : R₄⁻ → L₁₁, L₁₂
- L₁₁ : HALT
- L₁₂ : R₀⁺ → L₁₃
- L₁₃ : R₂⁻ → L₁₄, L₁₅
- L₁₄ : R₁⁺ → L₁₃
- L₁₅ : R₃⁻ → L₁₆, L₂
- L₁₆ : R₂⁺ → L₁₅



Alternatively, we could repeatedly divide the number, until it becomes 0.

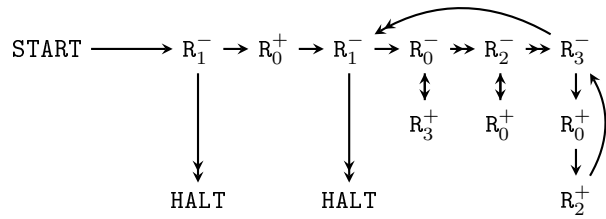
Exercise 23 [Fibonacci Numbers] Show that the *Fibonacci number* function $\text{Fib} : \mathbb{N} \rightarrow \mathbb{N}$, where

$$\text{Fib}(n) \triangleq \begin{cases} \text{Fib}(n-1) + \text{Fib}(n-2) & \text{if } n \geq 2 \\ n & \text{otherwise} \end{cases}$$

is RM computable. Outline the construction of an RM program that computes this function.

You can define the RM program in terms of high-level constructs, as a simple while loop. Alternatively, you can directly implement it:

- $L_0 : R_1^- \rightarrow L_2, L_1$
- $L_1 : \text{HALT}$
- $L_2 : R_0^+ \rightarrow L_3$
- $L_3 : R_1^- \rightarrow L_5, L_4$
- $L_4 : \text{HALT}$
- $L_5 : R_0^- \rightarrow L_6, L_7$
- $L_6 : R_3^+ \rightarrow L_5$
- $L_7 : R_2^- \rightarrow L_8, L_9$
- $L_8 : R_0^+ \rightarrow L_7$
- $L_9 : R_3^- \rightarrow L_{10}, L_3$
- $L_{10} : R_0^+ \rightarrow L_{11}$
- $L_{11} : R_2^+ \rightarrow L_9$

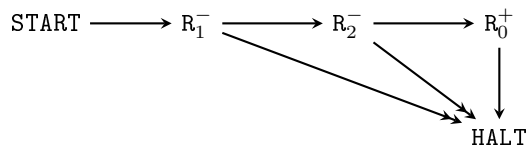


Exercise 24 [Boolean logic] Show that the binary logic functions **and**, **or** and **xor** are computable.

We represent true by 1 and false by 0. (There are different valid representations, like 0 is false and any $n > 0$ is true).

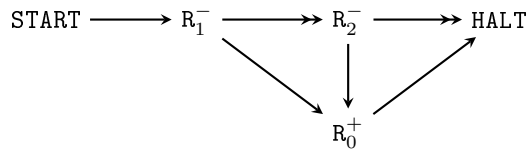
The **and** function is implemented as:

- $L_0 : R_1^- \rightarrow L_1, L_3$
- $L_1 : R_2^- \rightarrow L_2, L_3$
- $L_2 : R_0^+ \rightarrow L_3$
- $L_3 : \text{HALT}$



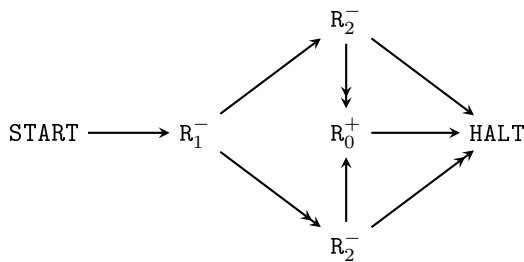
The **or** function is implemented as:

- $L_0 : R_1^- \rightarrow L_2, L_1$
- $L_1 : R_2^- \rightarrow L_2, L_3$
- $L_2 : R_0^+ \rightarrow L_3$
- $L_3 : \text{HALT}$



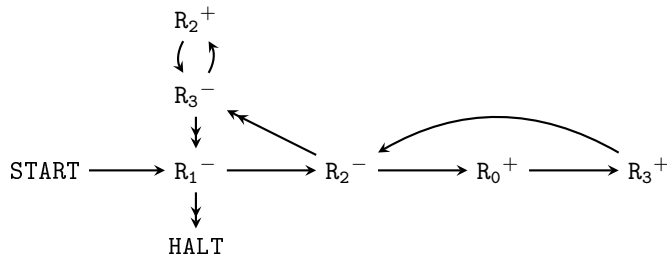
The **xor** function is implemented as:

- $L_0 : R_1^- \rightarrow L_1, L_3$
- $L_1 : R_2^- \rightarrow L_4, L_2$
- $L_2 : R_0^+ \rightarrow L_4$
- $L_3 : R_2^- \rightarrow L_2, L_4$
- $L_4 : \text{HALT}$



Exercise 25 [Reverse engineer the program 1] Attempt [2010P6Q3 (d)].

We start by creating the digram for this RM program:



The L_0 instruction is used to implement a while loop over R_1 . In each iteration R_2 is added to R_0 and to R_3 (so that it can restore R_2 in instructions L_4 and L_5). Hence, if initially $R_1 = x$ and $R_2 = y$, then after the execution of the program R_0 will store $x \cdot y$.

Exercise 26 [Reverse engineer the program 2] Attempt [1999P3Q9 (c)] (x' means x^+).

Assume initially that $S = 2^x(2y + 1)$ for some natural x and y . The first part zeros A and then checks if S is zero. If it is, we reach EXIT 0. Otherwise, S is copied to Z . Then, $S = Z/2$ (if Z is odd then it also terminates). Then A is incremented and S is moved to Z so that another iteration can be executed. This process repeats until Z is odd. At that point $A = x$ (number of times that division with 2 was successful) and $S = y$ (floor of the division of the remaining odd number with 2).

Lecture 3

Exercise 27 [Numerical codings of pairs]

- (a) Define $\langle\langle x, y \rangle\rangle$.
- (b) Why is this a bijection between $\mathbb{N} \times \mathbb{N}$ and $\mathbb{N} \setminus \{0\}$?
- (c) Show that the encoding and decoding functions are computable. (See [2017P6Q3 (a)(i),(ii)])
- (d) Define $\langle x, y \rangle$.
- (e) Why is this a bijection between $\mathbb{N} \times \mathbb{N}$ and \mathbb{N} ?
- (f) Show that the encoding and decoding functions are computable.

(a) $\langle\langle x, y \rangle\rangle \triangleq 2^x(2y + 1)$.

See Lecture 3 slide 8.

- (b) The binary representation of the result is given by $y | 1 | \underbrace{0 \dots 0}_x$. The function is injective because assuming that (x_1, y_1) and (x_2, y_2) map to the same value $v = y | 1 | \underbrace{0 \dots 0}_x$, then $x = x_1 = x_2$ (as the number of trailing zeros must be equal) and $y = y_1 = y_2$ (as the remaining digits must be equal).

For surjectivity, consider a value v that is non-zero. Since it is non-zero we can find the least significant 1 in its binary representation at position ℓ so $v = z \mid 1 \mid \underbrace{0 \dots 0}_{\ell}$. For $x = \ell$ and $y = z$, we get $\langle\langle\ell, z\rangle\rangle = v$.

Hence, the function is bijective.

See Lecture 3 slide 9.

(c) Look at Exercise 28 for encoding and at Exercise 26 for decoding.

(d) $\langle x, y \rangle \triangleq 2^x(2y + 1) - 1$

See Lecture 3 slide 8.

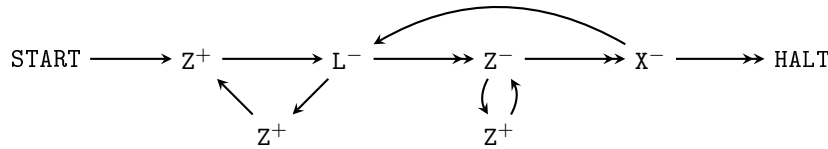
(e) The function is still injective because $x = y \Leftrightarrow x - 1 = y - 1$. For $\langle\langle\cdot, \cdot\rangle\rangle$ because it is surjective we have that for every value $v > 0$, there exists a pair (x, y) such that $v = \langle\langle x, y \rangle\rangle \Rightarrow v - 1 = \langle\langle x, y \rangle\rangle - 1 = \langle x, y \rangle$. Hence, for every $v \geq 0$, there exists a pair (x, y) such that $\langle x, y \rangle = v$. Hence, the function is bijective.

See Lecture 3 slide 9.

(f) For encoding we just compute $\langle\langle x, y \rangle\rangle$ and subtract 1 (which will always result in taking the non-zero branch). For decoding, we increment the value by 1 and then decode using $\langle\langle x, y \rangle\rangle$.

Exercise 28 [Reverse engineer the program 3] Attempt [2006P3Q7 (a)].

The program has the following diagram:



If the program starts with $X = 0$, $L = l$ and $Z = 0$, then when it halts: $X = 0$, $L = 2^x(2l + 1)$ and $Z = 0$. The first time X^- is reached the program, $L = 2l + 1$ and $Z = 0$. Then in each iteration $Z = 2L$ and then Z is moved to L . There are x iterations, so this has the effect of multiplying by 2^x .

Exercise 29 [Numerical codings of lists]

- (a) Show how a list is encoded using the numerical codings of pairs.
- (b) Is this encoding bijective?
- (c) Show that the encoding and decoding are computable.
- (d) (optional) Show how you could obtain an injective mapping for OCaml datatypes.
- (e) (optional) Show how to implement the for-each construct.

(a) For $\ell \in \text{list}\mathbb{N}$, define $\ulcorner \ell \urcorner \in \mathbb{N}$ by induction on the length of the list ℓ :

$$\begin{cases} \ulcorner \ell \urcorner & \triangleq 0 \\ \ulcorner x :: \ell \urcorner & \langle\langle x, \ulcorner \ell \urcorner \rangle\rangle = 2^x(2 \cdot \ulcorner \ell \urcorner + 1) \end{cases}$$

$$\text{or } \ulcorner [x_1; x_2; \dots; x_n] \urcorner = \langle\langle x_1, \langle\langle x_2, \dots \langle\langle x_n, 0 \rangle\rangle, \dots \rangle\rangle \rangle.$$

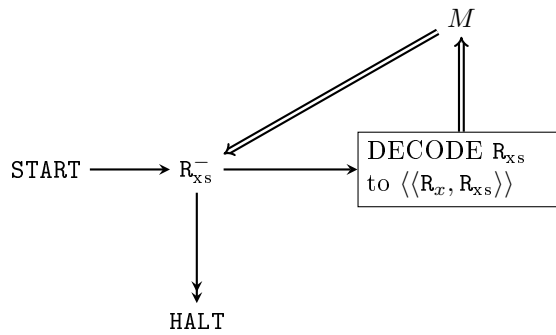
See Lecture 3 slide 12.

(b) We can prove by induction that if the list is $\ell = [x_1; x_2; \dots; x_n]$, then $\ulcorner \ell \urcorner = 1 \underbrace{0 \dots 0}_{x_1 \text{ zeros}} 1 \underbrace{0 \dots 0}_{x_2 \text{ zeros}} \dots 1 \underbrace{0 \dots 0}_{x_n \text{ zeros}} 0$.

Hence, two encodings are equal iff the length sequences of zeros are equal (injective). For surjectivity, we can construct a list of naturals, by counting the number of zeros between consecutive ones.

See Lecture 3 slide 16.

- (c) The encoding is computable because $\langle\langle\cdot, \cdot\rangle\rangle$ is computable.
- (d) Assume that we have a datatype that has constructors A_i (for $i = 1, \dots, k$) and takes ℓ_k arguments t_1, \dots, t_{ℓ_k} . Then we can encode it as $\ulcorner i; \ulcorner t_1 \urcorner; \ulcorner t_2 \urcorner; \dots \ulcorner t_{\ell_k} \urcorner \urcorner$. *Note:* The function need not be bijective.
- (e) Assume that the list of interest is placed in R_{xs} and the body of the for-each loop is implemented by M which uses the current element x of the list in register R_x .



Exercise 30 [Instruction encodings]

- (a) Explain how RM instructions are encoded.
- (b) How can these be decoded?
- (c) Show that both the encoding and decoding functions are computable.
- (d) Is this encoding a bijection?

(a) There are three types of instructions, so we just have to find a mapping for each. In the lectures, you used the following mapping:

$$\begin{cases} \ulcorner R_i^+ \urcorner \rightarrow L_j \urcorner & \triangleq \langle\langle 2i, j \rangle\rangle \\ \ulcorner R_i^- \urcorner \rightarrow L_j, L_k \urcorner & \triangleq \langle\langle 2i + 1, \langle j, k \rangle \rangle\rangle \\ \ulcorner \text{HALT} \urcorner & \triangleq 0 \end{cases}$$

See **Lecture 3 slide 17.**

(b) An encoding v can be decoded as follows: (1) check if v is zero (if yes, the HALT), (2) decode the $\langle\langle a, b \rangle\rangle$ and (i) if a is odd then $a = 2i + 1$ and decode b using $\langle j, k \rangle$ and (ii) if a is even, then b is the jump label.

See **Lecture 3 slide 18.**

- (c) The encoding and decoding functions are computable since $\langle\langle \cdot, \cdot \rangle\rangle$ and $\langle \cdot, \cdot \rangle$ have computable encodings and decodings.
- (d) Yes, since $\langle\langle \cdot, \cdot \rangle\rangle$ is and HALT occupies 0.

Exercise 31 [Program encodings]

- (a) How is an RM program encoded?
- (b) Why did the lecturer choose to have erroneous halts as well as proper halts?

(a) A RM program is just a list of instructions. Since we have a mapped instructions to naturals, we can just encode the list of naturals.

See **Lecture 3 slide 17.**

(b) Erroneous halts are allowed so that the encoding of programs is bijective.

Exercise 32

- (a) Attempt **[2014P6Q3 (a), (b)]**.
- (b) Read the statements for **[2011P6Q3 (a)]** and **[1996P3Q9 (a)]**.

See **solution notes**.

Exercise 33 [Program encoding in the emulator] (optional)

- (a) Write a function for the RM emulator that takes a program and returns the index of the program.
- (b) Write a function that takes the index of a program and returns the program.
- (c) Choose an index of your preference and check if the program terminates. (be careful with erroneous halts)

(d) See also [1999P3Q9 (d)], [1996P3Q9 (a)] or [1995P3Q9 (a)].

Exercise 34 [Counting programs] Attempt the following subquestions from [2007P3Q7 (b)(iii)]:

- (a) Explain why there are only countably many computable functions from $\mathbb{N} \rightarrow \mathbb{N}$.
- (b) Deduce that there exists a partial function from $\mathbb{N} \rightarrow \mathbb{N}$ that is not computable. (Any standard results you use about countable and uncountable sets should be clearly stated, but need not be proved.)
- (c) If a partial function f from $\mathbb{N} \rightarrow \mathbb{N}$ is computable, how many different register machine programs are there that compute f ?

- (a) In Lecture 3, you have seen how to obtain a bijection between natural numbers and RM programs. Hence, RM programs are countable and so are RM computable partial functions (since each computable partial function is computed by at least one RM).
- (b) From last year's Discrete Maths course, the powerset of \mathbb{N} is uncountable. But each subset S of \mathbb{N} defines a partial function in the following way: for $x \in S$ set $f(x) = 1$ and $f(x) = 0$ otherwise. Hence, there are at least as many partial functions (and so uncountable). Hence, there must exist partial functions that are not computable.
- (c) There are infinitely many. we can just append HALT instructions to the program list.

Lecture 4

Exercise 35 [Universal RM]

- (a) Define what is the *universal RM*. (See [2019P6Q5 (a)], [2013P6Q3 (a)], [2007P3Q7 (a) (ii)] or [1995P3Q9 (b)])
- (b) Describe the high level steps for creating a universal machine. (See [1999P3Q9 (e)])
- (c) (optional) Implement the universal RM in the emulator. Having implemented the program concatenation and the function that permutes the registers used by a program, will make your code much simpler.

- (a) A universal register machine U carries out the following computation, starting with $R_0 = 0$, $R_1 = e$ (code of a program), $R_2 = a$ (code of a list of arguments) and all other registers zeroed:
 - decode e as a program P
 - decode a as a list of register values a_1, \dots, a_n
 - carry out the computation of the program P starting with $R_0 = 0, R_1 = a_1, \dots, R_n = a_n$ (and any other registers occurring in P set to 0).

See Lecture 3 slide 25.

See Lecture 3 slide 27.

(b)

Exercise 36 [Number of computation steps] Explain how you would use the universal register machine construction to create RMs for the following:

- (a) Show that the function

$$\ell(e, x) = \begin{cases} \text{number of steps in the computation of } e(x) & \text{if } e(x) \text{ halts} \\ \text{undef} & \text{otherwise} \end{cases}$$

is computable. (See [2013P6Q3 (c) (iv)])

(b) Given two programs e_1 and e_2 , show how to create a program e , such that

$$e(x) = \begin{cases} \text{undef} & \text{if } \ell(e_1, x) = \ell(e_2, x) = \infty \\ e_1(x) & \text{if } \ell(e_1, x) \leq \ell(e_2, x) \\ e_2(x) & \text{otherwise} \end{cases}$$

(c) Given a program e and a natural t , write a function

$$\text{stops}(e, x, t) = \begin{cases} 1 & \text{if } \ell(e, x) \leq t \\ 0 & \text{otherwise} \end{cases}$$

- (a) We can add one more register C to the universal machine, so that just before the execution we increment C . Then we replace the halting instruction by zeroing the output register and moving there the contents of C . Hence, if the program terminates it outputs the number of steps that the machine executes.
- (b) We can have a universal machine that interleaves the execution of two RM programs, i.e. executes one instruction from $e_1(x)$ and one instruction from $e_2(x)$. When a halting configuration is encountered for e_1 then we output $e_1(x)$ and halt. When a halting configuration is encountered for e_2 we do the same for $e_2(x)$.
- (c) We modify the universal machine so that it keeps an additional register C that counts the number of instructions executed up to that point. Each time that we simulate an instruction we increment the counter. If a halt occurs, then we output 1 and halt. Otherwise, we check if the counter is equal to t . If it is, then we output 0 and halt.