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 mistakes 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.

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).

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

See Lecture 1 slide 15

See Lecture 1 slide 19

Lecture 2

Exercise 3

(a) Define a register machine (RM). (See [2010P6Q3 (a)] or [2007P3Q7 (a)(i)]

(b) Define a register machine configuration. (See [2009P6Q3 (a)] or [1999P3Q9 (b)]

(c) 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?

(d) What are the two ways that an RM halts?

(e) How can you modify a program to turn all erroneous halts into proper halts?

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

\[
L_0 : \text{body}_0, L_1 : \text{body}_1, \ldots, L_n : \text{body}_n
\]

where each \( \text{body}_i \) takes one of three possible forms:

- \( R_i^+ \rightarrow \text{L} \): add 1 to the contents of the \( i \)th register and then jump to the instruction labelled \( L \)
- \( R_i^- \rightarrow \text{L}, L' \): 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 \( L' \)
- \( \text{HALT} \): stop executing instructions.

See Lecture 2 slide 5
(b) A configuration \((\ell, [r_0, r_1, \ldots, r_n])\) of a register machine consists of the label \(\ell\) of the current program instruction together with a list \([r_0, r_1, \ldots, r_n]\) of the current contents of the machine’s registers \(R_0, R_1, \ldots, R_n\). See Lecture 2 slide 17.

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

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

By deterministic, we mean that for a given program, a given configuration and a given instruction \(\ell\), there is a unique next configuration. See Lecture 2 slide 18.

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

- either \(\ell^{th}\) instruction in program has body \texttt{HALT} (a “proper halt”)
- or \(\ell\) is greater than the number of instructions in program, so that there is no instruction labelled \(L_\ell\) (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 \texttt{HALT} instruction at the end of the program. This means that erroneous halts will be converted to proper \texttt{HALTS} at instruction \(L\).

(Alternative- a bit more wasteful), append \texttt{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:

<table>
<thead>
<tr>
<th>instruction</th>
<th>representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(R^+ \rightarrow L)</td>
<td>(R^+ \rightarrow [L])</td>
</tr>
<tr>
<td>(R^- \rightarrow L, L')</td>
<td>(R^- \rightarrow [L], [L'])</td>
</tr>
<tr>
<td>\texttt{HALT}</td>
<td>\texttt{HALT}</td>
</tr>
<tr>
<td>(L_0)</td>
<td>\texttt{START} (\rightarrow [L_0])</td>
</tr>
</tbody>
</table>

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 \(X \leftarrow Y\).

(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
\[
\forall x \in X. \forall y \in Y. (x, y) \in f \land (x, y') \in f \implies y = y'.
\]
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) \(\hat{f}(x) \downarrow \iff \exists y \in Y. f(x) = y\),
\(\hat{f}(x) \downarrow \iff \neg \exists y \in Y. f(x) = y\),
\[X \rightarrow Y := \{f \mid f\text{ a total function}\},\]
\[X \leftrightarrow 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

\[
\text{START} \longrightarrow \text{HALT}
\]

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

\[
\text{START} \longrightarrow \text{R}_0
\]

**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) \(f(x_1, \ldots, x_n)\) is register machine computable if there is a register machine \(M\) with at least \(n + 1\) registers such that for all \((x_1, \ldots, x_n) \in \mathbb{N}^n\) and all \(y \in \mathbb{N}\), the computation of \(M\) starting with \(R_0 = 0, R_1 = x_1, \ldots, R_n = x_n\) and all other registers zeroed halts with \(R_0 = y\) if and only if \(f\) is defined at \((x_1, \ldots, x_n)\) and takes value \(y\) there.
See Lecture 2 slide 34

(b) The I/O convention states that registers \(R_1, \ldots, R_n\) 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).

```java
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)),
```

3
SUB(R(3), L(5), L(0)),
ADD(R(2), L(4)),
HALT
), doNotAllowErroreousHalts);
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=3;R3=1;])
(1, [R0=2;R1=1;R2=3;R3=1;])
(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=3;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=3;R3=1;])
(2, [R0=4;R1=0;R2=3;R3=1;])
(3, [R0=5;R1=0;R2=3;R3=1;])
(1, [R0=5;R1=0;R2=3;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=3;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;])

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 \(\Rightarrow\) notation in RM diagrams, where \(M \Rightarrow M'\) 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:

\[
\text{START} \quad \xrightarrow{} \quad M_1 \quad \xrightarrow{} \quad M_2 \quad \xrightarrow{} \quad \text{HALT}
\]

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:

\[
\text{START} \quad \xrightarrow{} \quad M_1 \quad \xrightarrow{} \quad R^- \quad \xrightarrow{} \quad M_2 \quad \xrightarrow{} \quad \text{HALT}
\]

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:
(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 \( \text{for}(M_1; M_2; M_3)\{M_4\} \) 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 \( p_1^2 : \mathbb{N}^2 \to \mathbb{N} \), 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:

\[
\begin{align*}
L_0 &\quad: R_1^- \to L_1, L_2 \\
L_1 &\quad: R_0^+ \to L_0 \\
L_2 &\quad: \text{HALT}
\end{align*}
\]

For the second projection function, we change \( 1 \to 2 \):

\[
\begin{align*}
L_0 &\quad: R_2^- \to L_1, L_2 \\
L_1 &\quad: R_0^+ \to L_0 \\
L_2 &\quad: \text{HALT}
\end{align*}
\]

**Exercise 11 [Constant]** Show that the constant function \( c : \mathbb{N} \to \mathbb{N} \), 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 \( \text{add} : \mathbb{N}^2 \to \mathbb{N} \), where \( \text{add}(x, y) \defeq x + y \) is RM computable. Write the RM program and create a diagram for this.

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

**Exercise 14 [Max]** Show that the *max* function \( \text{max} : \mathbb{N}^2 \to \mathbb{N} \) 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 \( t_{\text{sub}} : \mathbb{N}^2 \to \mathbb{N} \), where

\[
t_{\text{sub}}(x, y) \triangleq \begin{cases} 
  x - y & \text{if } y \leq x \\
  0 & \text{if } y > x
\end{cases}
\]

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

**Exercise 16 [Comp]** Show that the comp function \( \text{comp} : \mathbb{N}^2 \to \mathbb{N} \), 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 \( t_{\text{sub}} \) and then check if it is positive (and output 1) otherwise halt.

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

Any RM that loops forever is good here:

\[
L_0 : R_0^+ \to L_0
\]

**Exercise 18 [Integer division]** Show that the integer division function \( \text{div} : \mathbb{N}^2 \to \mathbb{N} \), 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.
Exercise 19 [Mod] Show that the mod (basically $\lfloor \cdot \rfloor$ as defined in Discrete Maths) function $\text{mod} : \mathbb{N}^2 \to \mathbb{N}$, 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:

---

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

---
**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).

```
START 
Z := 1 
WHILE R_2 > 0: 
  T := Z \cdot R_0 
  Z := T 
HALT
```

**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.

```
L_0 : R_1^- \rightarrow L_2, L_1 
L_1 : \text{HALT} 
L_2 : R_2^+ \rightarrow L_3 
L_3 : R_2^- \rightarrow L_4, L_7 
L_4 : R_3^+ \rightarrow L_5 
L_5 : R_3^- \rightarrow L_6 
L_6 : R_4^+ \rightarrow L_3 
L_7 : R_1^- \rightarrow L_8, L_{10} 
L_8 : R_4^+ \rightarrow L_9 
L_9 : R_4^- \rightarrow L_7, L_7 
L_{10} : R_4^- \rightarrow L_{11}, L_{12} 
L_{11} : \text{HALT} 
L_{12} : R_0^+ \rightarrow L_{13} 
L_{13} : R_2^- \rightarrow L_{14}, L_{15} 
L_{14} : R_1^+ \rightarrow L_{13} 
L_{15} : R_3^- \rightarrow L_{16}, L_2 
L_{16} : R_2^+ \rightarrow L_{15}
```

Alternatively, we could repeatedly divide the number, until it becomes 0.
**Exercise 23 [Fibonacci Numbers]** Show that the *Fibonacci number* function $Fib : \mathbb{N} \rightarrow \mathbb{N}$, where

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

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:

$$\begin{align*}
L_0 : & \ R^- \rightarrow L_2, L_1 \\
L_1 : & \ \text{HALT} \\
L_2 : & \ R^+ \rightarrow L_3 \\
L_3 : & \ R^- \rightarrow L_5, L_4 \\
L_4 : & \ \text{HALT} \\
L_5 : & \ R^0 \rightarrow L_6, L_7 \\
L_6 : & \ R^+ \rightarrow L_5 \\
L_7 : & \ R^2 \rightarrow L_8, L_9 \\
L_8 : & \ R^0 \rightarrow L_7 \\
L_9 : & \ R^3 \rightarrow L_{10}, L_3 \\
L_{10} : & \ R^+ \rightarrow L_{11} \\
L_{11} : & \ R^+ \rightarrow L_9 \\
\end{align*}$$

**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:

$$\begin{align*}
L_0 : & \ R^- \rightarrow L_1, L_3 \\
L_1 : & \ R^- \rightarrow L_2, L_3 \\
L_2 : & \ R^+ \rightarrow L_3 \\
L_3 : & \ \text{HALT} \\
\end{align*}$$

The *or* function is implemented as:

$$\begin{align*}
L_0 : & \ R^- \rightarrow L_1, L_1 \\
L_1 : & \ R^- \rightarrow L_2, L_3 \\
L_2 : & \ R^+ \rightarrow L_3 \\
L_3 : & \ \text{HALT} \\
\end{align*}$$

The *xor* function is implemented as:
Exercise 25 [Reverse engineer the program 1] Attempt 2010P6Q3 (d).

We start by creating the digram for this RM program:

```
START  →  R₁⁻  ↓  R₂⁺  ↓  R₃⁻  ↓  R₄⁻  →  R₂⁻  →  R₅⁺  →  HALT
```

The L₀ instruction is used to implement a while loop over R₁. In each iteration R₂ is added to R₀ and to R₃ (so that it can restore R₂ in instructions L₄ and L₅). Hence, if initially R₁ = x and R₂ = y, then after the execution of the program R₀ will store x · y.

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

Assume initially that S = 2x(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 ⟨⟨x, y⟩⟩.
(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 ⟨x, y⟩.
(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) \( ⟨⟨x, y⟩⟩ ≜ 2^x(2y + 1) \).

(b) The binary representation of the result is given by \( y | 1 | 0 \ldots \ 0 \) \( x \) zeros. The function is injective because assuming that \((x₁, y₁)\) and \((x₂, y₂)\) map to the same value \( v = y | 1 | 0 \ldots 0 \) \( x \) zeros, then \( x = x₁ = x₂ \) (as the number of trailing zeros must be equal) and \( y = y₁ = y₂ \) (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 \( \ell \) so \( v = z_1 \ldots 0_\ell \). For \( x = \ell \) and \( y = z \), we get \( \langle \langle \ell, z \rangle \rangle = v \).

Hence, the function is bijective.

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

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

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

(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 \([2006P3Q7 \text{ (a)}]\).

The program has the following diagram:

```
START -> Z+ -> L^- -> Z^- -> X^- -> HALT

Z+ -> Z^- -> L-

X^- -> Z^+
```

If the program starts with \( X = 0 \), \( L = l \) and \( Z = 0 \), then when it halts: \( X = 0 \), \( L = 2^l(2l + 1) \) 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 to 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 \( \ell \in \text{list} N \), define \( \Gamma \ell \in N \) by induction on the length of the list \( \ell \):

\[
\begin{align*}
\Gamma \ell & \triangleq 0 \\
\Gamma x :: \ell & \triangleq 2^x(2 \cdot \Gamma \ell + 1)
\end{align*}
\]

or \( \Gamma [x_1; x_2; \ldots; x_n] \) = \( \langle \langle x_1, \langle x_2, \ldots, \langle x_n, 0 \rangle \rangle \rangle \rangle \).

See \([Lecture 3 slide 12]\).

(b) We can prove by induction that if the list is \( \ell = [x_1; x_2; \ldots; x_n] \), then \( \Gamma \ell = 10 \ldots 0 1 0 \ldots 0 \ldots 1 0 \ldots 0 \).

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 \( i = 1, \ldots, k \)) and takes \( \ell_k \) arguments \( t_1, \ldots, t_{\ell_k} \). Then we can encode it as \( \Gamma i; \Gamma t_1; \Gamma t_2; \ldots; \Gamma t_{\ell_k} \). Note: The function need not be injective.

(e) Assume that the list of interest is placed in \( R_x \) and the body of the for-each loop is implemented by \( M \) which uses the current element \( x \) of the list in register \( R_x \).

13
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:

\[
\begin{align*}
R_i^+ \rightarrow L_j & \triangleq \langle \langle 2i, j \rangle \rangle \\
R_i^- \rightarrow L_j, L_k & \triangleq \langle \langle 2i + 1, \langle j, k \rangle \rangle \rangle \\
\text{HALT} & \triangleq 0
\end{align*}
\]

See Lecture 3 slide 17.

(b) An encoding \( v \) can be decoded as follows: (1) check if \( v \) is zero (if yes, the \text{HALT}), (2) decode the \( \langle a, b \rangle \) 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.

(c) The encoding and decoding functions are computable since \( \langle \cdot, \cdot \rangle \) and \( \cdot, \cdot \rangle \) have computable encodings and decodings.

(d) Yes, since \( \langle \cdot, \cdot \rangle \) is and \text{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.

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

(a) Explain why there are only countably many computable functions from \( \mathbb{N} \to \mathbb{N} \).

(b) Deduce that there exists a partial function from \( \mathbb{N} \to \mathbb{N} \) 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 \( \mathbb{N} \to \mathbb{N} \) 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 \texttt{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 \( a_1, \ldots, a_n \)
- carry out the computation of the program \( P \) starting with \( R_0 = 0, R_1 = a_1, \ldots , R_n = a_n \) (and any other registers occurring in \( P \) set to 0).

(b)

Exercise 36 [Number of computation steps] Explain how you would use the universal register machine construction to create RMIs 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 the 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.