1 Example Sheet

1. Show that the following arithmetic functions are all register machine computable.

   Note: I've implemented and tested these all in C using macros and `goto` to simulate a register machine.

   (a) First projection function $p \in \mathbb{N}^2 \to \mathbb{N}$, where $p(x, y) \triangleq x$:

   $$\begin{aligned} L_0 &: X^- \to L_1, L_2 \\ L_1 &: R_0^+ \to L_0 \\ L_2 &: \texttt{HALT} \end{aligned}$$

   (b) Constant function with value $n \in \mathbb{N}$, $c \in \mathbb{N} \to \mathbb{N}$, where $c(x) \triangleq n$

   $$\begin{aligned} L_0 &: R_0^+ \to L_1 \\ L_1 &: R_0^+ \to L_2 \\ &\vdots \\ L_{n-1} &: R_0^+ \to L_n \\ L_n &: \texttt{HALT} \end{aligned}$$

   (c) Truncated subtraction function, $\_-\_ \in \mathbb{N}^2 \to \mathbb{N}$, where $x - y \triangleq \begin{cases} x - y & \text{if } y \leq x \\ 0 & \text{otherwise} \end{cases}$

   $$\begin{aligned} L_0 &: Y^- \to L_1, L_2 \\ L_1 &: X^- \to L_0, L_4 \\ L_2 &: X^- \to L_3, L_4 \\ L_3 &: R_0^+ \to L_2 \\ L_4 &: \texttt{HALT} \end{aligned}$$

   (d) Integer division function, $\_div\_ \in \mathbb{N}^2 \to \mathbb{N}$, where

   $$x \; div \; y \triangleq \begin{cases} \left\lfloor \frac{x}{y} \right\rfloor & \text{if } y > 0 \\ 0 & \text{if } y = 0 \end{cases}$$

   $$\begin{aligned} L_0 &: Y^- \to L_1, L_7 \\ L_1 &: X^- \to L_2, L_7 \\ L_2 &: Z^+ \to L_3 \\ L_3 &: Y^- \to L_1, L_4 \\ L_4 &: R_0^+ \to L_5 \\ L_5 &: Z^- \to L_6, L_0 \\ L_6 &: Y^+ \to L_5 \\ L_7 &: \texttt{HALT} \end{aligned}$$

sorry, I can't read those,
better stick with graphs in the exam :)

(e) Integer remainder function, $\_mod\_ \in \mathbb{N}^2 \to \mathbb{N}$, where $x \ mod \ y \triangleq x - y(x \ div \ y)$

$$
\begin{aligned}
L_0 &: Y^- \to L_1, L_{13} \\
L_1 &: Y^+ \to L_2 \\
L_2 &: X^- \to L_3, L_5 \\
L_3 &: R_0^+ \to L_4 \\
L_4 &: Z^+ \to L_2 \\
L_5 &: Z^- \to L_6, L_7 \\
L_6 &: X^+ \to L_5 \\
L_7 &: Y^- \to L_8, L_{10} \\
L_8 &: Z^+ \to L_9 \\
L_9 &: X^- \to L_7, L_{13} \\
L_{10} &: Z^- \to L_{11}, L_{12} \\
L_{11} &: Y^+ \to L_{10} \\
L_{12} &: R_0^- \to L_{12}, L_2 \\
L_{13} &: \texttt{HALT}
\end{aligned}
$$

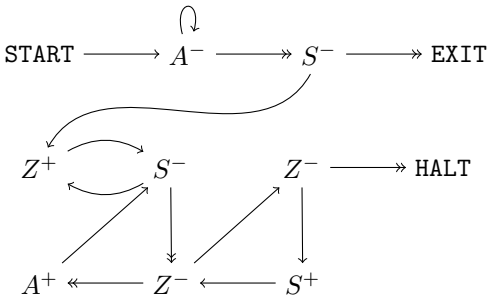
(f) Exponentiation base 2, $e \in \mathbb{N} \to \mathbb{N}$, where $e(x) \triangleq 2^x$.

$$
\begin{aligned}
L_0 &: R_0^+ \to L_1 \\
L_1 &: X^- \to L_2, L_7 \\
L_2 &: R_0^- \to L_3, L_5 \\
L_3 &: Z^+ \to L_4 \\
L_4 &: Z^+ \to L_2 \\
L_5 &: Z^- \to L_6, L_1 \\
L_6 &: R_0^+ \to L_5 \\
L_7 &: \texttt{HALT}
\end{aligned}
$$

(g) Logarithm base 2, $\lg \in \mathbb{N} \to \mathbb{N}$, where $\lg(x) \triangleq \begin{cases} y \ s.t. \ 2^y \leq x < 2^{y+1} & \text{if } x > 0 \\ 0 & \text{if } x = 0 \end{cases}$

$$
\begin{aligned}
L_0 &: X^- \to L_1, L_3 \\
L_1 &: X^- \to L_2, L_3 \\
L_2 &: Z^+ \to L_0 \\
L_3 &: Z^- \to L_4, L_7 \\
L_4 &: R_0^+ \to L_5 \\
L_5 &: X^+ \to L_6 \\
L_6 &: Z^- \to L_5, L_0 \\
L_7 &: \texttt{HALT}
\end{aligned}
$$

3. Consider the list of register machine instructions whose graphical representation is shown below. Assuming that register `Z` holds 0 initially, describe what happens when the code is executed (both in terms of the effect on registers `A` and `S` and whether the code halts by jumping to the label `EXIT` or `HALT`).



POP from the lecture notes

The register machine removes all multiples of 2 from S and stores the number of multiples of 2 it has removed in A. The program halts by jumping to `EXIT` if and only if $S = 0$. Otherwise, it halts by jumping to `HALT`.

More Formally:

which one is which?

$$M(A,S) = \begin{cases} (\texttt{HALT}, n, m) & \text{if } (A,S) = (k, m \cdot 2^n) \\ (\texttt{EXIT}, 0, 0) & \text{if } (A,S) = (k, 0) \end{cases}$$

Pop S to A

2 2003 Paper 3 Question 7

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2003p3q7.pdf

(a) What is meant by a *register machine*? Explain the action of a register machine program.

A register machine is a theoretical machine used as a model of computation. It consists of a finite list of instructions and a finite list of registers containing natural numbers.

There are three types of instructions:

Increment increases the value of register R and jumps to instruction with label L_j

$$L_i : R^+ \to L_j$$

If R is nonzero, Decrement will decrease R and jump to the instruction with label L_j; else do not decrement R and jump to instruction L_k

$$L_i : R^- \to L_j, L_k$$

`HALT` terminates the program

$$L_i : \texttt{HALT}$$

A register machine starts at the instruction with label L_0 with n arguments in registers $R_1, \ldots, R_n$. It then executes the instructions as specified.

A register machine terminates either by executing a `HALT` or erroneously by accessing a nonexistent register or jumping to a nonexistent instruction.

Register machines with enough registers are turing complete. Therefore whether an arbitrary Register machine halts is undecidable in general.

(b) What does it mean for a partial function $f(x_1, \ldots, x_n)$ of n arguments to be *register machine computable*?

A partial function $f(x_1, \ldots, x_n)$ is register machine computable if and only if there exists a register machine M with at least $n+1$ registers which when run with arguments $R_1 = x_1, \ldots, R_n = x_n$ and all other registers set to 0 terminates with $R_0 = y$ if and only if $f(x_1, \ldots, x_n) = y$.

(c) Design register machines to compute the following functions.

(i) $f(x_1, x_2) = x_1 + x_2$

$$
\begin{aligned}
L_0 &: X_1^- \to L_1, L_2 \\
L_1 &: R_0^+ \to L_0 \\
L_2 &: X_2^- \to L_3, L_4 \\
L_3 &: R_0^+ \to L_2 \\
L_4 &: \texttt{HALT}
\end{aligned}
$$

(ii) $g(x_1) = \begin{cases} 42 & \text{if } x + 1 > 0 \\ \text{undefined} & \text{otherwise} \end{cases}$

$$
\begin{aligned}
L_0 &: X_1^- \to L_2, L_1 \\
L_1 &: R_0^+ \to L_1 \\
L_2 &: R_0^+ \to L_3 \\
L_3 &: R_0^+ \to L_4 \\
&\vdots \\
L_{44} &: R_0^+ \to L_{45} \\
L_{45} &: \texttt{HALT}
\end{aligned}
$$

(iii) $h(x_1) = 2^{x_1}$

Already done in Exercise Sheet, Exercise 1, Part (f)

(d) Give an example of a function that is not register machine computable, stating clearly any well-known results you use.

$$f(e, x) = \begin{cases} 1 & \text{if the } e(x) \downarrow \\ 0 & \text{otherwise} \end{cases}$$

If a register machine M for the function f existed, then it would solve the halting problem. It is a standard result that there exists no machine which can solve the halting problem. Therefore the register machine M cannot exist.

3 Universal Register Machine

(a) What is a Universal Register Machine? What does it do?

A Universal Register Machine is a register machine which can simulate other register machines; including itself. It takes as input a representation of other register machines and an integer representation of the contents of their registers.

A Universal Register Machine takes two arguments:

- e – an encoded representation of the program of the register machine it is simulating
- a – an encoded representation of the initial state of the registers of the register machine it is simulating.

(b) How does it work? Draw an implementation of URM and mark separate parts.

A Universal Register Machine takes two arguments: an integer representation of another register machine and an integer representation of that register machines registers.

Coding Programs as Numbers

The arguments of the Universal Register Machine must be integer representations of programs.

Define the two following ways of making a pair:

$$\langle\langle x, y\rangle\rangle \triangleq 2^x \left(2 \cdot y + 1\right) \tag{1}$$

$$\langle x, y\rangle \triangleq 2^x \left(2 \cdot y + 1\right) - 1 \tag{2}$$

$$\tag{3}$$

Define lists by $[] = 0$ and $x :: xs = \langle x, xs\rangle$. For example

$$[x_2, x_1, x_0] = \langle\langle x_2 \langle\langle x_1, \langle\langle x_0, 0\rangle\rangle\rangle\rangle\rangle\rangle$$

Define the following correspondences:

$$L_w : R_x^+ \to L_y \qquad \simeq \langle\langle 2 \cdot x, y\rangle\rangle \tag{4}$$

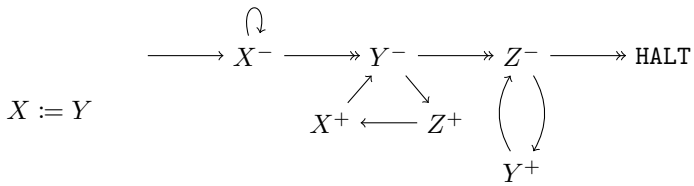
$$L_w : R_x^- \to L_y, L_z \qquad \simeq \langle\langle 2 \cdot x + 1, \langle y, z\rangle\rangle\rangle \tag{5}$$

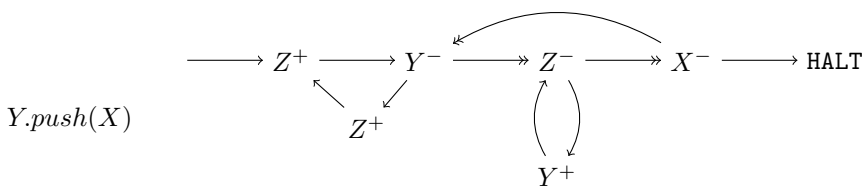
Register machine programs are lists of instructions. We have an injection between instructions and integers; and between integer lists and integers; therefore we also have an injection between instruction lists and integers. So we have in injection between register machine programs and integers.

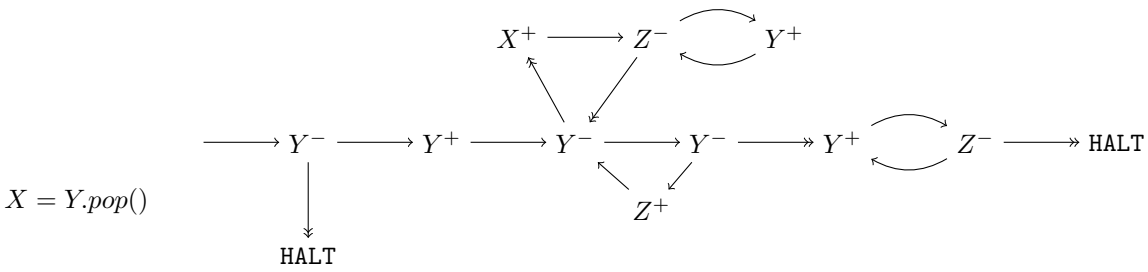
We can also represent the initial state of the registers using the injection from integer lists to integers.

Universal Register Machine

Using the following macros:



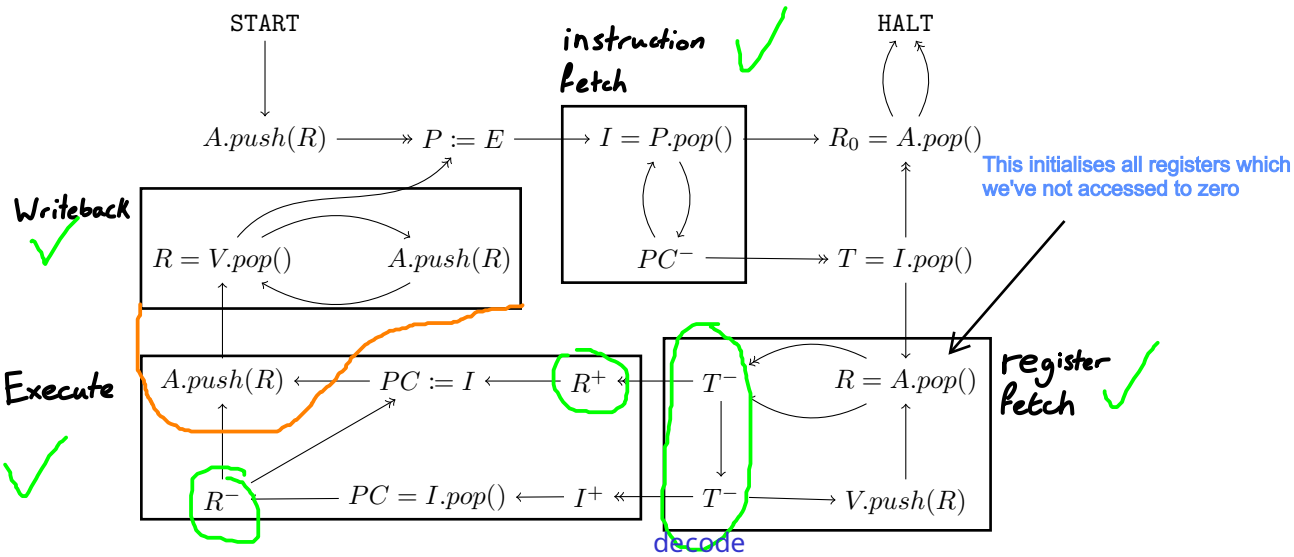




With the registers of the Universal register machine initialised as follows:

$R_0 = 0$	Return value
$E = e$	Program to be simulated
$A = R$	Arguments of the simulated machine
$PC = 0$	Program Counter
$I = 0$	Instruction currently being simulated
$T = 0$	Type of the current instruction
$R = 0$	Value of the register being manipulated
$V = 0$	List of lower registers than the register being processed
$P = 0$	Copy of the program to be simulated

Using the macros above, the following register machine is a Universal Register Machine:



(c) Look at your diagram again. Does it remind you of something? Explain the similarities.

The diagram of a Universal Register Machine is like a Unicore CPU with no parallelism.

Similarities:

- Both take integer representations of programs as arguments.
- Both are used to simulate higher-level "virtual machines"
- Both operate on registers

Differences:

- CPUs have resource constraints while URMs do not
- CPUs support higher-level operations such as multiplication and division, while URMs do not. This reduces the computational complexity of many common tasks.
- (x86) CPUs have types, while URMs do not
- CPUs exploit instruction level parallelism, while URMs do not.

4 2007 Paper 3 Question 7

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2007p3q7.pdf

(a) (i) Define the notion of a *register machine* and the computations that it carries out.

A register machine is a theoretical machine which is used as a model of computation. Register machines have a finite number of registers of infinite precision. Register machine instructions operate on registers. Register machines support three primitive operations:

- $L_i : R_j^+ \to L_k$

  Increment R_j then go to instruction L_k
- $L_i : R_j^- \to L_k, L_l$

  If R_j is zero then go to instruction L_l else decrement R_j and go to instruction L_k.
- `HALT`

  Halts execution.

Register machine programs are lists of instructions. They can be "passed arguments" by initialising the values of certain registers. After computation finishes, the return value is considered to be the value in register R_0.

(ii) Explain, in general terms, what is meant by a *universal register machine*. (You should make clear what scheme for coding problems as numbers you are using, but you are not required to describe a universal register machine program in detail.)

A Universal Register Machine is a register machine which can simulate execution or any other register machine, including itself.

Universal Register Machines take two arguments – an integer representation of the program which it simulates e and an integer representation of the arguments on which that program is run a.

A suitable coding scheme is as follows:

$$\langle\langle x, y\rangle\rangle = 2^x \cdot (2 \cdot y + 1)$$
$$\langle x, y\rangle = 2^x \cdot (2 \cdot y + 1) - 1$$

Instructions can be defined as follows:

$$\begin{aligned} L : R_i^+ \to L_j &= \langle\langle 2 \cdot i, j\rangle\rangle \\ L : R_i^- \to L_j, L_k &= \langle\langle 2 \cdot i + 1, \langle j, k\rangle\rangle\rangle \\ L : \texttt{HALT} &= 0 \end{aligned}$$

Lists can be encoded as follows:

$$[\,] = 0$$
$$x :: y = \langle\langle x, y\rangle\rangle$$

The argument e to a Universal register machine can be a list of instructions defined as above. The argument a can be represented as a list of integers where $a[i]$ is the initial state of register R_i.

(b) (i) Explain what it means for a partial function f from $\mathbb{N} \to \mathbb{N}$ to be computable by a register machine.

A partial function $f : \mathbb{N} \rightharpoonup \mathbb{N}$ is register machine computable iff there exists a register machine M with at least 2 registers such that when M is run with $R_1 = x$ and all other registers 0; M will terminate with $R_0 = y$ if and only if $f(x) = y$.

(ii) Let $n > 1$ be a fixed natural number. Show that the partial function from $\mathbb{N} \to \mathbb{N}$

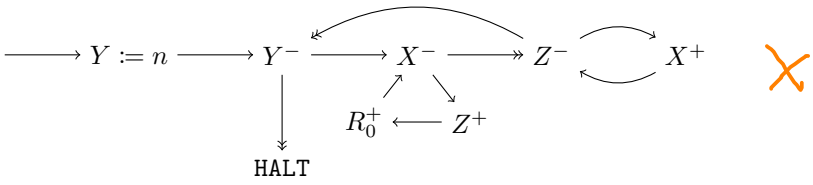
$$f_n(x) = \begin{cases} nx & \text{if } x > 0 \\ \text{undefined} & \text{if } x = 0 \end{cases}$$

is computable.

Define the macro $X := n$ for some constant n as:

$$\longrightarrow X^+ \longrightarrow X^+ \longrightarrow \cdots \longrightarrow X^+ \longrightarrow$$

The following register machine computes the function f_n (with argument X and return register R_0). Since there exists a register machine which computes the function f_n, f_n is computable.



(iii) Explain why there are only countably many computable functions from $\mathbb{N} \to \mathbb{N}$ Deduce that there exists a partial function $\mathbb{N} \to \mathbb{N}$ that is not computable. (Any standard results you use about countable and uncountable sets should be clearly stated, but need not be proved.)

Partial functions $\mathbb{N} \rightharpoonup \mathbb{N}$ are equivalent to elements f in the powerset $\mathcal{P}\left(\mathbb{N}^2\right)$. The cardinality of the set $\mathbb{N} \times \mathbb{N}$ is $\aleph_0$.

yes but that wasn't defined in lectures

The cardinality of the powerset of q set of cardinality $\aleph_0$ is $\aleph_1$. Therefore, the number of partial functions from integers to integers is uncountably infinite.

For all computable functions, there exists a register machine program which computes them. Since all register machine programs can all be represented as integers; for all computable functions there exists an integer which represents them. Therefore the cardinality of the set of computable functions is less than or equal to $\aleph_0$.

you might want to mention encodings and bijections/ surjections that we defined earlier

We can easily provide examples of sets of computable partial functions of cardinality $\aleph_0$ ($\{\{(i,i)\}|i \in \mathbb{N}\}$) – which proves by example that the cardinality of the set of computable functions is greater or equal to $\aleph_0$.

Combining the two above results, we can conclude that the cardinality of the set of computable functions is $\aleph_0$.

By simple cardinalities we can conclude that the cardinality of the set of uncountable functions $\mathbb{N} \rightharpoonup \mathbb{N}$ is $\aleph_1 - \aleph_0 = \aleph_1$. Therefore, there are infinitely many uncomputable partial functions $\mathbb{N} \rightharpoonup \mathbb{N}$.

(iv) If a partial function f from $\mathbb{N} \rightharpoonup \mathbb{N}$ is computable, how many different register machine programs are there that compute f?

If $f : \mathbb{N} \rightharpoonup \mathbb{N}$ is computable then there exists a register machine M which computes it. We can use M to define another register machine which also computes f as below.

$$\longrightarrow R_0^+ \longrightarrow R_0^- \rightleftarrows M$$

New machines can be created by this method infinitely. Therefore, there are infinitely many register machines which compute f. Since we can represent all

good point

register machines as integers, there are countably infinitely many register machines. Therefore, we can conclude that the number of register machines which compute f is countably infinite: $\aleph_0$.

5 2014 Paper 6 Question 3

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2014p6q3.pdf

(a) Explain how to code register machine programs P as numbers $\ulcorner P \urcorner \in \mathbb{N}$ so that each $e \in \mathbb{N}$ can be decoded to a unique register machine program $\text{prog}(e)$.

A register machine program is a list of instructions. So if we can encode both instructions and a list, we can encode a register machine as an integer.

Define:

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

Next, inductively define a list:

$$\ulcorner [\,] \urcorner \triangleq 0$$
$$x :: xs \triangleq \langle\langle x, \ulcorner xs \urcorner\rangle\rangle$$

Define the instructions:

$$\ulcorner \texttt{HALT} \urcorner \triangleq 0$$
$$\ulcorner R_i^+ \to L_j \urcorner \triangleq \langle\langle 2 \cdot i, j\rangle\rangle$$
$$\ulcorner R_i^- \to L_j, L_k \urcorner \triangleq \langle\langle 2 \cdot i + 1, \langle j, k\rangle\rangle\rangle$$

A list $[x_0, x_1, \ldots, x_n]$ has the following binary representation:

$$\underbrace{11\ldots11}_{x_0}0\underbrace{11\ldots11}_{x_1}0\ldots0\underbrace{11\ldots11}_{x_n}$$

isn't it the other way around?
1(0000: x_n zeros)1(00..: x_{n-1} zeros)1...

This is clearly a bijection between integers and lists. [3]=3::[]=<<3,0>>=2^3(2*0+1)=8 = 0b1000

We can use this to encode and decode programs (the encoding of which is a bijection between integers and programs).

(b) Find a number $e_1 \in \mathbb{N}$ for which $prog(e_1)$ is a register machine program for computing the function $one \in \mathbb{N} \to \mathbb{N}$ with $one(x) = 1$ for all $x \in \mathbb{N}$

The register machine one has the below code:

$$L_0 : R_0^+ \to L_1$$
$$L_1 : \texttt{HALT}$$

We must first convert each instruction to an integer, then place them in a list.

let's discuss

$$R_0^+ \to L_1 = \langle 2 \cdot 0, 1\rangle = \langle 0, 1\rangle = 1 \quad (6)$$
$$\texttt{HALT} = 0 \quad (7)$$
$$e_1 = [R_0^+ \to L_1; \texttt{HALT}] = [1, 0] = \langle 1, 0\rangle = 3 \quad (8)$$

[1,0] = 1::(0::[]) = <<1, <<0, 0>>>>

Therefore, the integer representation of the program $prog$ under this encoding is 3

(c) Why is it important for the theory of computation that the functions involved in the coding and decoding given in part (a) are themselves register machine computable?

For a register machine to be a useful model of computation, it must be Turing Complete. A necessary condition for this is that a Universal Register Machine must exist. For this to be possible, there must be an encoding which represents programs that the register machine can use. Halting problem

(d) Define what it means for a set of numbers $S \subseteq \mathbb{N}$ to be register machine *decidable*.

A set of numbers $S \subseteq \mathbb{N}$ is register machine *decidable* if there exists a register machine M which can determine whether or not a given number is in the set S.

(e) Let $\varphi_e \in \mathbb{N} \rightharpoonup \mathbb{N}$ denote the partial function of one argument computed by the register machine with program $prog(e)$. Prove that $\{e \in \mathbb{N} \mid \varphi_e = one\}$ is register machine undecidable (where *one* is the function mentioned in part (b)). State carefully any results you use in your proof.

Let S denote the set $\{e \in \mathbb{N} \mid \varphi_e = one\}$.

Take an arbitrary integer i. Define e as the integer such that $prog(e)$ is equivalent to $prog(i)$ but with all occurrences of `HALT` replaced by a program setting R_0 to 1. Therefore $\phi_e = one$ if and only if $prog(i)$ halts on all inputs.

The function from i to e is computable. If membership of the set S was computable, we could solve the Halting Problem by calculating e for a given program i and checking whether $e \in S$. It's a standard result that there exists no program which solves the halting problem. Therefore, by contradiction membership of the set S must not be computable.

let's discuss this,
in Halting problem, we're
given program AND input

6 Implementing a Register Machine

(a) Implement a register machine in OCaml or C++. No need for the perfect implementation.

To minimise the semantic difference: here are three C macros which can be used to write a basic register machine. An example program is also shown.

```
typedef unsigned long long ull;
#define P(X, Y) X++; goto Y;
#define S(X, Y, Z) if (X == 0) {goto Z;} else {X--; goto Y;}
#define H return r0;

int logarithm(int x){
    ull r0=0, t=0;
    l0: S(x, l1, l3)
    l1: S(x, l2, l3)
    l2: P(t, l0)
    l3: S(t, l4, l7)
    l4: P(r0, l5)
    l5: P(x, l6)
    l6: S(t, l5, l0)
    l7: H
}
```

COOL :)

(b) State what simplifications or assumptions you made (or would have made). Discuss how those simplifications affect algorithms that your register machine can execute. Is it equivalent to the real register machine? If not, is the halting decidable in your case? If yes, sketch an algorithm to do that.

This register machine has finite integer size. Therefore our program is equivalent to a DFA – the state is given by $L \times \times R_1 \times \cdots \times R_n$ where $R_i \in \mathbb{N}_{<2^{64}}$. Halting is decidable for DFAs and therefore the halting problem is decidable for our machine.

In a real register machine, the domain of values in each register R_i is infinite. So the number of states is countably infinite. However, in our machine, integers have 2^{64} possible states. So it is possible to record which states in the DFA we've been to and return that a program never halts if it ever-reenters a state. However this algorithm is computationally impossible. For example the above program, with 8 lines and 3 registers has 2^{195} states – the algorithm would require 5.85×10^{48}Gb to run.

Due to finite integer size, we cannot use strategies such as unary encoding beyond small examples – a common paradigm on register machines.

works in simple cases, e.g. in stack trace we see same arguments again and again

(c) Is it possible to write a C++ compiler that runs on your virtual machine? What limitations would it have?

It is possible to write a C++ compiler that runs on the virtual machine. Since the machine supports only very basic operations, the computational complexity would be entirely impractical.

(d) Highlight a few similarities and differences between a popular programming language of your choice and a register machine.

I will compare a register machine to Python:

Similarities:

- Both languages are Turing Complete
- Both register machines and Python support infinite precision integers

Differences:

- Python has variable scoping, register machine programs do not support this concept – however, it could be simulated with a suitable calling convention.
- Python has exceptions – this concept does not exist in register machine programs.
- Python has complex datatypes. Register machines support only the infinite precision integer datatype.
- Python has higher-level primitive operations than register machine – for example addition of two variables is constant complexity, unlike register machines where the complexity of $x + y$ is $\mathcal{O}(x + y)$.
- Python can take input from users, register machines cannot.
- Python provides support for interfacing with other programming languages – register machines do not – ie there is no way to call a function written on a Turing Machine from a register machine.
- Parsing Python programs is context-sensitive, while parsing register machine programs is context-free.
- Register machines support `GOTO`, while Python does not.

Most important point: in programming languages we usually rely on ``infinite'' memory to keep all of our data, while in RMs we encode lists and other data structures into infinite numbers. The point of all these encodings that we were playing around is to show that these things are equivalent. Of course, you already knew that: information in the computer is stored in binary, so everything is just one big number :).

All the examples about reducing a problem to the halting problem always make a change to the input and pass it to this new program
Is it also possible to modify just the output or modify both of it?

This is absolutely possible
But, its a much less powerful construction.
We want to create a machine that solves the halting problem
This machine has a very specific input.
The idea of the proof is to mangle the inputs that are halting problme solvable to trick the other machine into solving the halting problem.
Inputs to our machine are fixed -- we want to adjust them to pass it to the other problem we assume exists.

All non-trivial static analysers of Turing complete program languages cannot possibly tell.
Arithmetic is undecidable -- so they can't even check whether an array access will be out of bounds.
Normally, we work with fixed size integers so this becomes checkable.

More interesting problems of programs are almost always undecidable
Almost all analysers have to make compromises and either give false positives or false negatives.
We cannot have a total programming language (All valid programs terminate) that is Turing Complete.
It's infeasible to automatically check that a program terminates in total programming languages

If you have an unproven maths problem
In C, you can have a program which goes through the values
In a Total programming language, you cannot do this!

Arithmetic is undecidable (Entscheidungsproblem) -- this is the problem
If you find an undecidable problem, you can add it as an axiom

Include that all other registers are set to zero in the definition of whether a function is register machine computable

For/while loops have loops in the graph of the register machine

In classical programming, we have infinite set of finite integers
In register machines, we have finite set of infinite integers

Look or at least glance through the lecture notes the day before the exam

Use the graphical representations of register mahcines in the exams

You can use earlier register machines -- but its often easier just to draw the whole diagram

Specify explicitly what arguments are

We define erroneous halts because of how we encode programs
We don't check that all jumps or instructinos are within some range
We just define them as "halt" -- this makes our encoding a bijection -- so whatever program we encode, the program is valid
This avoids any nasty manipulation

We could also say that a machine only has registers which are used for input or in the program.
We then could not have an invalid program that used a register which didn't exist.

A specific register machine is like a specific program
PROGRAMS of register machines are Turing Complete

IMPORTANT:
Don't forget -- if a partial function is undefined at a value then the register machine which computes it should loop forever at this point

Bijection between register machines and integers
Surjection between register machines and computable partial functions
Use Cantors Diagonalization to prove that there exist partial funcitons which cannot be mapped to by the integers

You can modify a computable function by adding a halt onto the end of the list of instructions.

A decideable subset is a subset of the natural numbers for which implements the characteristic function for this set.

IE "determine but guarantee termination"

For the Halting Problem we are given both the program AND the inputs on which it is run!

Don't forget the encoding symbol!