CST Part IB

COMPUTATION THEORY

Andrew Pitts

Corrections to notes & extra material available from course web page:

www.cl.cam.ac.uk/teaching/0809/CompTheory

Introduction: algorithmically undecidable problems

There are inherent limitations on what [mathematical] problems can be solved using computers. Even with the idealization that the amount of time & working space available to carry out a computation is unlimited, there exist problems that are computationally unsolvable.

Three famous examples sketched in this lecture:

- Hilbert's Entscheidungsproblem
- The Halting Problem
- Hilbert's 10th Problem

Hilbert's Entscheidungsproblem

Is there an algorithm [aka "effective procedure"] which when fed any statement in the formal language of first order arithmetic, determines in a finite number of operations whether the statement is provable from Peano's axioms for arithmetic using the usual rules of classical logic ?

Posed by Hilbert at the 1928 International Congress of Mathematicians, and in fact the problem was stated in a more ambitious form, with a more powerful formal system in place of first order arithmetic.

The algorithm Hilbert's Entscheidungsproblem asks for would be a rather useful thing to have! E.g. we could run it on the statement

$$\forall k > 1 . \exists p, q . \ 2k = p + q \ \& \ \mathrm{prime}(p) \ \& \ \mathrm{prime}(q)$$

to find out whether Goldbach's Conjecture has a proof; etc., etc.

Hilbert believed that such an algorithm could be found. In 1930 he wrote

"In an effort to give an example of an unsolvable problem, the philosopher Comte once said that science would never succeed in ascertaining the secret of the chemical composition of the bodies of the universe. A few years later this problem was solved... The true reason, according to my thinking, why Comte could not find an unsolvable problem lies in the fact that there is no such thing as an unsolvable problem."

[quoted from C. Reid's biography of Hilbert]

A few years later Hilbert was proved wrong, by Church and Turing's work of 1935-36.

'Entscheidungsproblem' means 'decision problem'.

General form of a **decision problem** specified by:

- set S whose elements are finite datastructures of some kind (eg: formulas in formal system for first order arithmetic)

  infinite, if the problem is to be non-trivial

- property P of elements of S (eg: property of a formula that it has a proof)

The problem is then: find an algorithm A which

- always terminates with result 0 or 1 when fed an element $s \in S$, and
- $\underbrace{\text{yields result } 1 \text{ when fed } s}_{\text{write } A(s)=1 \text{ to indicate this}}$ if & only if s has prop. P

Algorithms, or "effective procedures"

No precise definition at time Hilbert posed the Entscheidungsproblem, just examples...

Common features of the examples:

- finite description of the procedure in terms of "elementary" operations
- deterministic (i.e. "next step" is uniquely determined, if there is one)
- procedure may not terminate on some input data, but can recognize when it does & what the result is

Examples of algorithms abound in the history of mathematics, eg:

- procedure for multiplying numbers in decimal notation
- procedure for extracting square roots to any desired accuracy
- procedure for finding highest common factors (Euclid's Algorithm)

etc., etc.

In 1935/36 Turing and Church gave independent, negative solutions of Hilbert's Entscheidungsproblem. The essential first step was to formulate a precise, mathematical definition of the notion of 'algorithm'.

Turing's formulation (in terms of what are now called Turing machines — of which more later) appeared more general/fundamental than Church's formulation (in terms of his lambda calculus), but Turing proved that both formulations described the same class of functions from [numerical] inputs to [numerical] outputs.

(Turing machines prefigured, and partly stimulated, the early development of digital computing. Lambda calculus partly inspired the development of 'functional' programming languages.)

Next step of Turing/Church solution of the Entscheidungsproblem: with a precise definition of 'algorithm', one can regard algorithms as data on which algorithms can act. Hence one can consider...

The Halting Problem

= the decision problem with

S = set of all pairs (A, D) where A is an algorithm and D is a datum on which it is designed to operate

P = property of such pairs given by:

"algorithm A when applied to datum D eventually produces a result (i.e. eventually halts)"

$$\underbrace{\text{``algorithm } A \text{ when applied to datum } D \text{ eventually produces a result (i.e. eventually halts)''}}_{\text{write } A(D)\downarrow \text{ to indicate this}}$$

write $A(D)\downarrow$ to indicate this

Turing and Church showed that the Halting Problem is undecidable, i.e. there is no algorithm H such that for all $(A, D) \in S$

$$H(A,D) = \begin{cases} 1 & \text{if } A(D)\downarrow \\ 0 & \text{otherwise} \end{cases}$$

Sketch of the proof

If there were such an H, we could use it to define an algorithm C:

"input A, compute $H(A,A)$ and if it is equal to 0 then return value 1 & halt, otherwise loop forever."

So for all A, $C(A)\downarrow \iff H(A,A)=0$ (since H total)
and for all A, $H(A,A)=0 \iff A(A)\not\downarrow$ (by def^n of H)
Hence for all A, $C(A)\downarrow \iff A(A)\not\downarrow$
Taking A to be C itself, we get $C(C)\downarrow \iff C(C)\not\downarrow$ contradiction!

The final step in the Turing/Church proof of the undecidability of Hilbert's Entscheidungsproblem was to show that the problem can be reduced to the Halting Problem, by constructing an algorithm for encoding instances (A, D) of the Halting Problem set as arithmetic statements $\Phi_{A,D}$ such that

$$\Phi_{A,D} \text{ is provable} \iff A(D)\downarrow$$

Hence any algorithm for deciding provability of arithmetic statements could be used to solve the Halting Problem — so no such can exist.

With hindsight, a positive solution to the Entscheidungsproblem would be too good to be true. However, the algorithmic unsolvability of some decision problems is much more surprising. A famous example of this is...

Hilbert's 10th Problem

Give an algorithm which, when started with any Diophantine equation, determines in a finite number of operations whether there are natural numbers satisfying the equation.

One of a number of important open problems listed by Hilbert at the International Congress of Mathematicians in 1900.

Diophantine equations

$$p(x_1,\dots,x_n) = q(x_1,\dots,x_n)$$

p, q polynomials in $x_1,\dots,x_n$ with coefficients from $\mathbb{N} = \{0,1,2,3,\dots\}$.

Named after Diophantus of Alexandria (c. 250 AD)

E.g. "Find three whole numbers (x, y, z) such that the product of any two added to the third is a square" [Diophantus' Arithmetica, Book III, Problem 7]

i.e. find $x, y, z \in \mathbb{N}$ for which there exists $u, v, w \in \mathbb{N}$

with $(xy + z - u^2)^2 + (yz + x - v^2)^2 + (zx + y - w^2)^2 = 0$

i.e. with $(x^2y^2 + y^2z^2 + z^2x^2 + \cdots) = (u^2xy + v^2yz + w^2zx + \cdots)$

[One solution: $(x,y,z) = (1,4,12)$, with $(u,v,w) = (4,7,4)$.]

Hilbert's 10th Problem was eventually shown to be reducible to the Halting Problem and hence algorithmically undecidable only in 1970 by the combined efforts of Y. Matijasevič, J. Robinson, M. Davis and H. Putnam. The original proof used Turing machines and was quite complicated. Later J.P. Jones & Y. Matijasevič (Jour. Symb. Logic 49(1984) 818–829) simplified a large part of the proof by using a formulation of algorithm (equivalent to Turing machines or λ-calculus) in terms of <u>register machines</u> – a formulation invented by Minsky & Lambek in the 1960s.

We will use register machines to develop the ideas sketched in this lecture and make them precise. We will return to Turing and Church's formulation of the notion of algorithm later in the course.

Register machines

Algorithms, or "effective procedures"

No precise definition at time Hilbert posed the Entscheidungsproblem, just examples...

Common features of the examples:

- finite description of the procedure in terms of "elementary" operations
- deterministic (i.e. "next step" is uniquely determined, if there is one)
- procedure may not terminate on some input data, but can recognize when it does & what the result is

Register Machines

The "elementary operations" for register machines will be :

- add 1 to a natural number
- test whether a nat.number is 0
- subtract 1 from a +ve nat. no.
- jump ("goto")
- Conditional ("if-then-else")

0, 1, 2, 3, ...
Stored in
(idealised)
registers

DEFINITION: a **register machine** consists of:

- finitely many **registers** R0, …, Rn
  (each capable of storing a natural number)
- a **program**, consisting of a finite list of **instructions** of the form

  label : body of instruction

— $(i+1)^{th}$ instruction in the list is labelled Li ($i = 0, 1, \dots$)

— instructions take one of three forms:

$L : R^+ \to L'$	add 1 to contents of register R and jump to instruction labelled L'
$L : R^- \to L', L''$	if contents of R is >0, subtract 1 from it and jump to L', otherwise jump to L''
$L : \mathrm{HALT}$	stop executing instructions

Example: register machine for addition

registers R0 R1 R2

program

$L0 : R1^{-} \to L1 , L2$

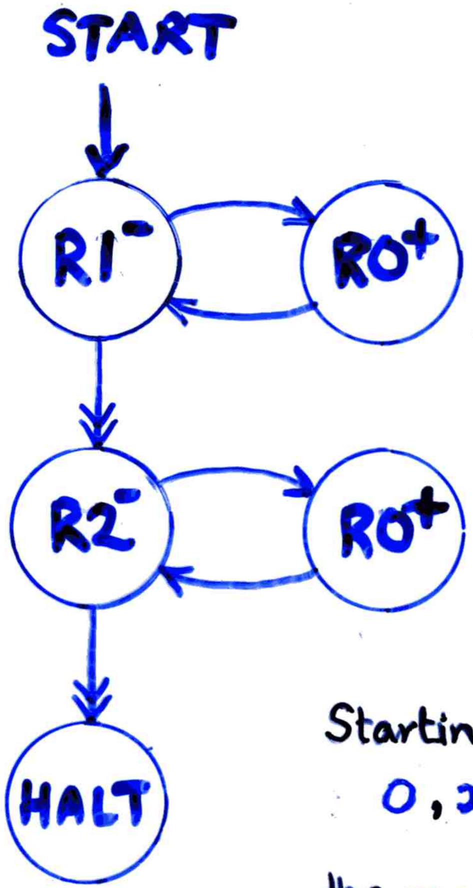
$L1 : R0^{+} \to L0$

$L2 : R2^{-} \to L3, L4$

$L3 : R0^{+} \to L2$

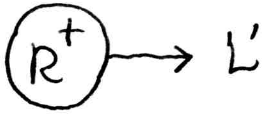
$L4 :$ HALT

graphical representation

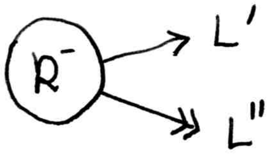


Starting with initial values $0, x, y$ in R0, R1, R2 the machine reaches L4: HALT with values $x+y, 0, 0$ in R0, R1, R2.

Graphical notation for programs

Increment instruction $L : R^{+} \to L'$ represented by 

Conditional decrement instruction $L : R^{-} \to L', L''$ represented by



Halt instruction represented by  .

So : - nodes of graph indicate register operations (& halting)

- arc of graph represent jumps between instructions

- labels of instructions become implicit

- loose sequential order of instructions, which is no problem so long as the START instruction of the graph is indicated.

Example: register machine for addition

registers R0 R1 R2

program

$L0: R1^- \to L1, L2$

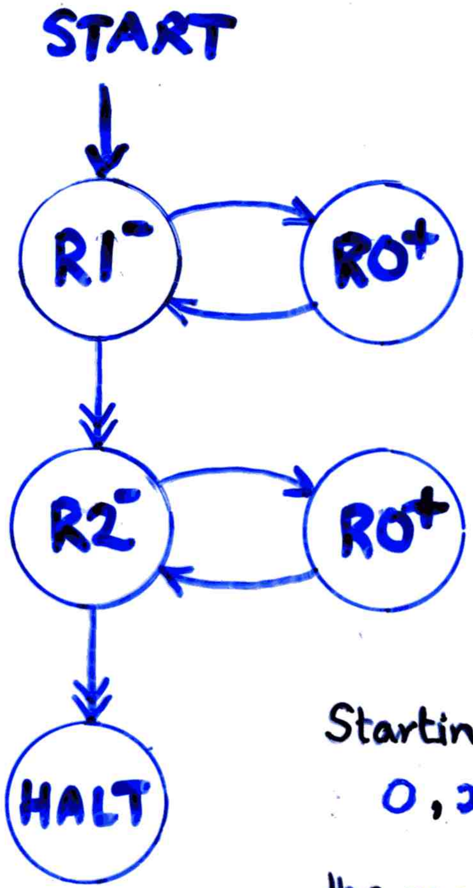
$L1: R0^+ \to L0$

$L2: R2^- \to L3, L4$

$L3: R0^+ \to L2$

$L4:$ HALT

graphical representation



Starting with initial values $0, x, y$ in R0, R1, R2 the machine reaches L4: HALT with values $x+y, 0, 0$ in R0, R1, R2.

Example : register machine for addition

registers R0 R1 R2

program

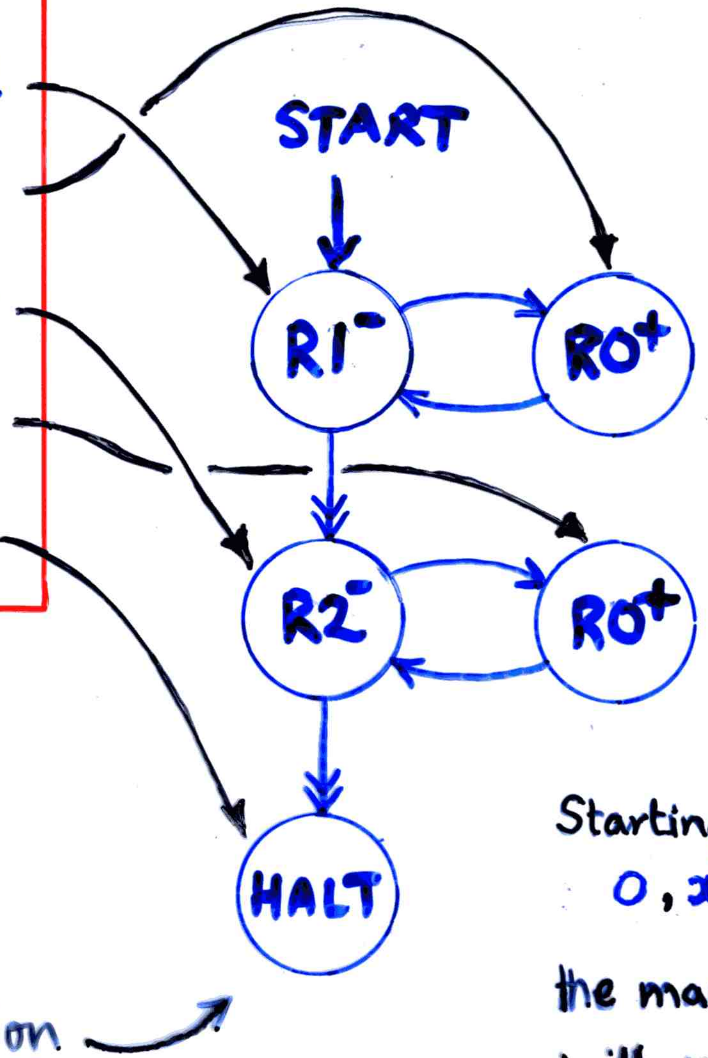
L0 : $R1^- \to$ L1 , L2

L1 : $R0^+ \to$ L0

L2 : $R2^- \to$ L3, L4

L3 : $R0^+ \to$ L2

L4 : HALT



graphical representation

Starting with initial values $0, x, y$ in R0, R1, R2 the machine reaches L4: HALT with values $x+y, 0, 0$ in R0, R1, R2 .

Example: register machine for addition

registers R0 R1 R2

program

$L0 : R1^- \to L1, L2$

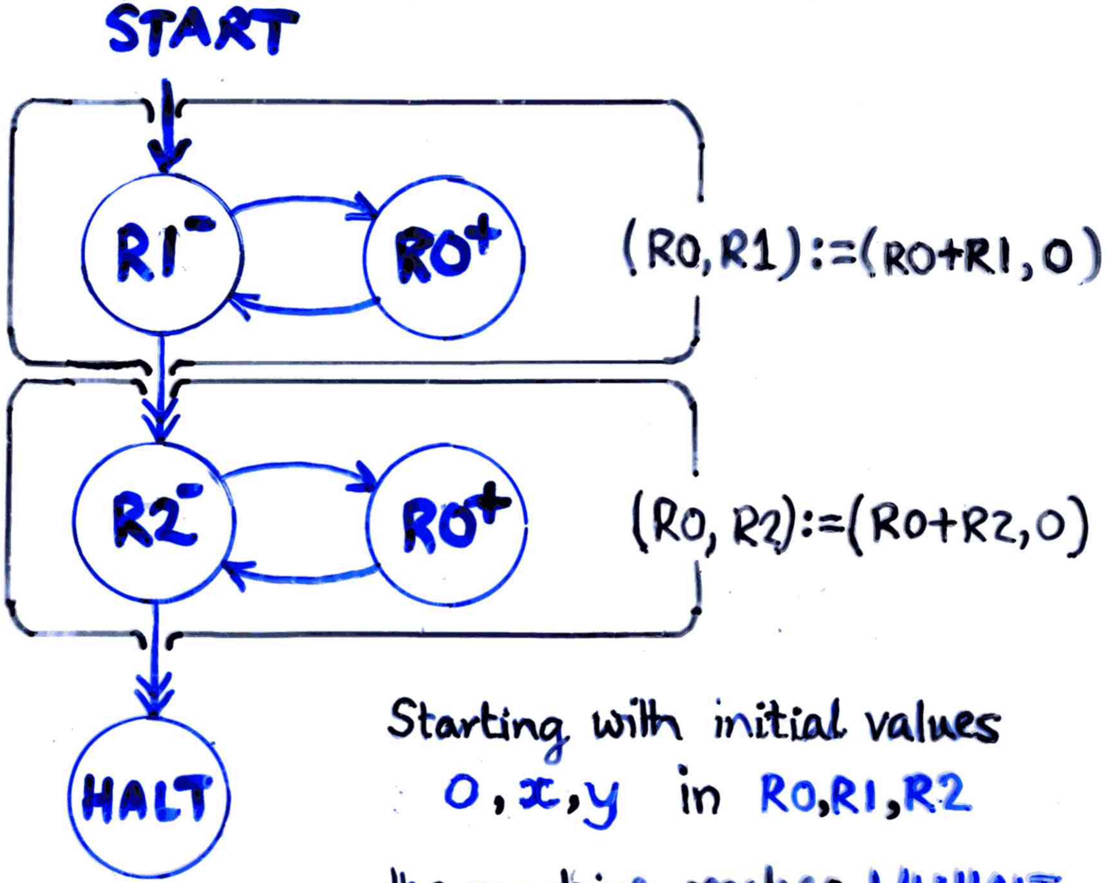
$L1 : R0^+ \to L0$

$L2 : R2^- \to L3, L4$

$L3 : R0^+ \to L2$

L4: HALT

graphical representation



Starting with initial values $0, x, y$ in R0, R1, R2

the machine reaches L4: HALT with values $x+y, 0, 0$ in R0, R1, R2.

Having loaded registers with initial values (from $\mathbb{N} = \{0,1,2,\ldots\}$)

> a **computation** (or **run**) of the register machine consists of obeying the program instructions, starting with the first in the list

EITHER

the computation continues forever ← e.g.

$L1 : R1^+ \to L1$
$L2 : \mathrm{HALT}$

never halts

OR

it halts, because

EITHER

a **HALT** instruction is obeyed
("proper halt")

OR

we are asked to jump to a label not in the list ← e.g.
("erroneous halt")

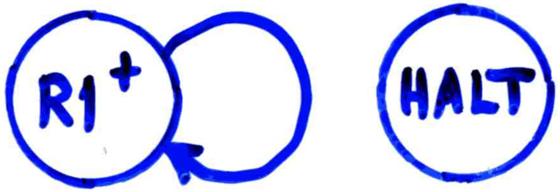
$L1 : R1^+ \to L3$
$L2 : \mathrm{HALT}$

halts erroneously

Graphical represtation of

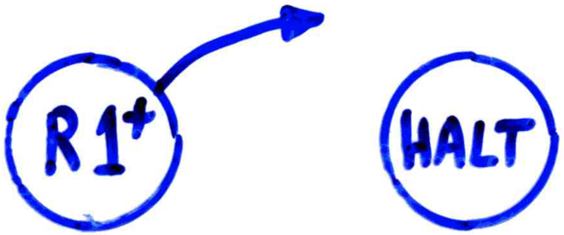
$L1 : R1^{+} \rightarrow L1$
$L2 : \text{HALT}$

is



$L1 : R1^{+} \rightarrow L3$
$L2 : \text{HALT}$

is



N.B. Can always modify programs to turn erroneous halts into proper halts by adding extra HALT instructions to the list with appropriate labels.

Note that the operation of a register machine is **deterministic**, in the sense that the next instruction to be obeyed (if any) is uniquely determined.

Because of this determinism and the possibility that computations do not halt, the relation between initial and final register contents defined by a register machine program is a **partial function** ...

DEFINITION : A partial function from a set X to a set Y is specified by any subset $f \subseteq X \times Y$ (of the set $X \times Y \overset{\text{def}}{=} \{(x,y) \mid x \in X \ \& \ y \in Y\}$ of ordered pairs) satisfying $(x,y) \in f \ \& \ (x,y') \in f \Rightarrow y = y'$

(i.e. for all $x \in X$ there is at most one $y \in Y$ with $(x,y) \in f$).

NOTATION :

$f(x) = y$ means $(x,y) \in f$

$f(x){\downarrow}$ means there is some y such that $(x,y) \in f$

$f(x){\uparrow}$ means there is no " " " "

$Pfn(X,Y) \overset{\text{def}}{=}$ set of all partial functions from X to Y

$Fun(X,Y) \overset{\text{def}}{=}$ set of all (total) functions from X to Y

↑

$\boxed{\text{those } f \in Pfn(X,Y) \text{ such that } f(x){\downarrow} \text{ for all } x \in X}$

DEFINITION :

$f \in \mathrm{Pfn}(\mathbb{N}^n, \mathbb{N})$ is (register machine) computable if & only if there is a register machine M with at least $n+1$ registers, $R0, R1, R2, \ldots, Rn$ say, (and maybe some other registers as well) with the property that for all $(x_1, \ldots, x_n) \in \mathbb{N}^n$ and all $y \in \mathbb{N}$

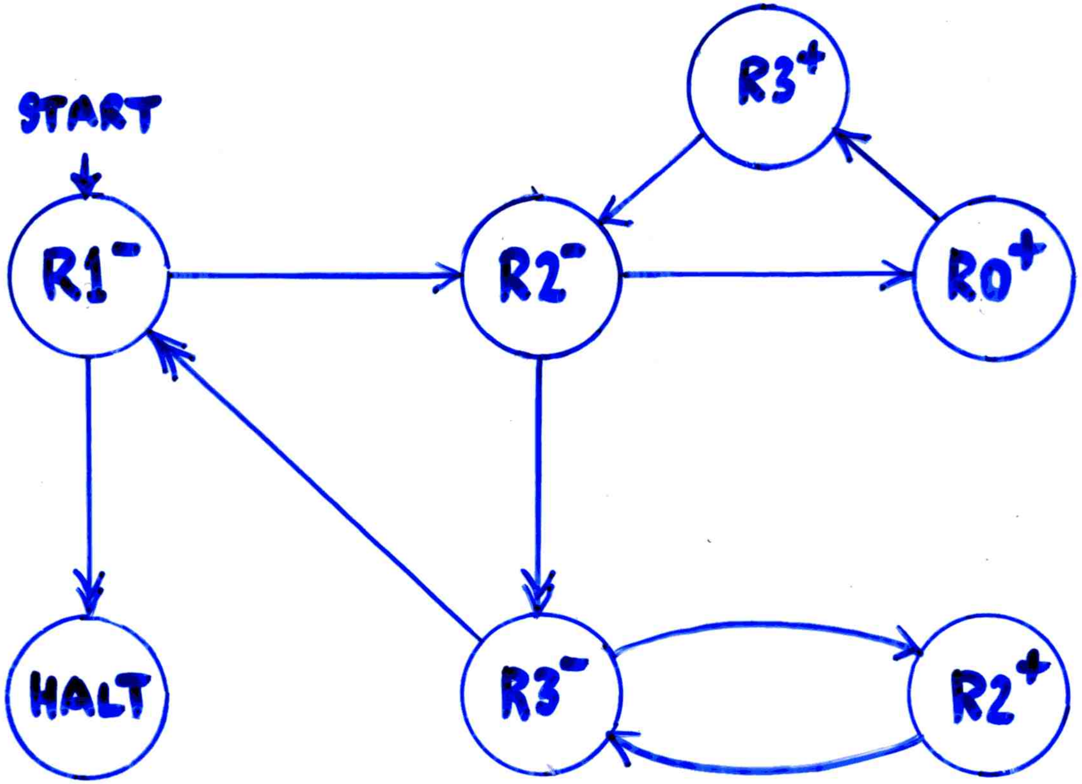
$f(x_1, \ldots, x_n) = y$ if & only if the computation of M starting with $R1 = x_1, \ldots, Rn = x_n$, and all other registers $= 0$, halts with $R0 = y$.

E.g. the example register machine on Slide 18 shows that the addition function $f(x,y) \stackrel{\text{def}}{=} x+y$ is computable.

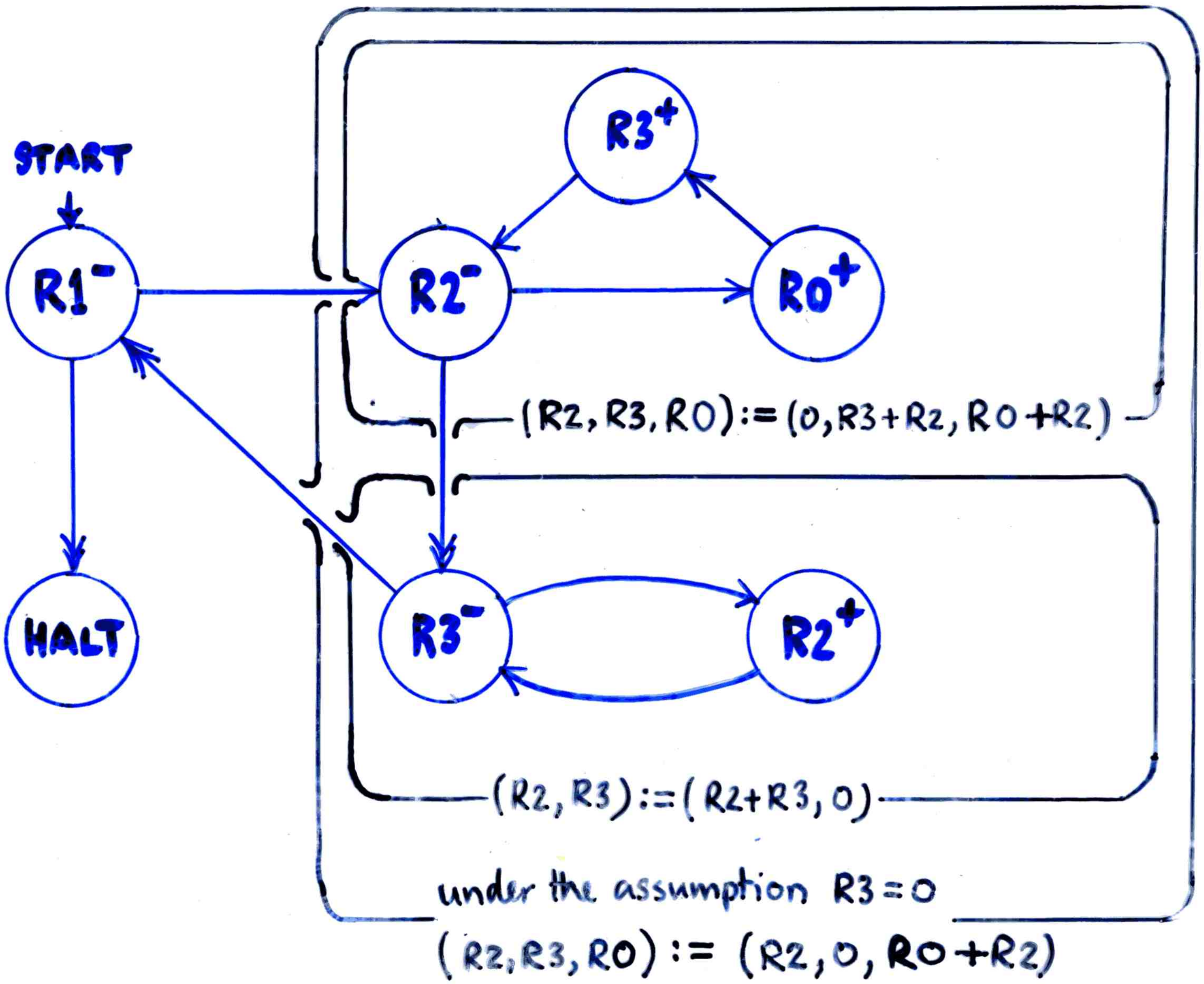
N.B. there may be many different register machines that compute the same partial function.

Of course the investigation of what kinds of function are computable, and what kinds are not, is a major concern of this course. For the moment let's just see some more examples...

Multiplication: $f(x,y) = xy$ is computable



Multiplication: $f(x,y) = xy$ is computable



Hence if the machine is started with $(R1, R2, R3, R0) := (x, y, 0, 0)$, it halts with $(R1, R2, R3, R0) := (0, y, 0, xy)$

Proposition.

The following arithmetic functions are all computable:

(1) projection $f(x,y) \overset{\text{def}}{=} x$

(2) constant $f(x) \overset{\text{def}}{=} n$

(3) truncated subtraction $x \dot{-} y \overset{\text{def}}{=} \begin{cases} x-y, & \text{if } y \leq x \\ 0, & \text{if } y > x \end{cases}$

(4) integer division $x \operatorname{div} y \overset{\text{def}}{=} \begin{cases} \text{integer part of } x/y, & \text{if } y > 0 \\ 0, & \text{if } y = 0 \end{cases}$

(5) integer remainder $x \bmod y \overset{\text{def}}{=} x \dot{-} y(x \operatorname{div} y)$

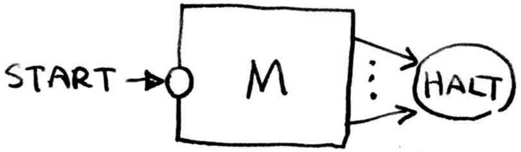
(6) exponentiation (base 2) $f(x) = 2^x$

(7) log base 2 $\log_2(x) \overset{\text{def}}{=} \begin{cases} \text{greatest } y \text{ s.t. } 2^y \leq x, & \text{if } x > 0 \\ 0, & \text{if } x = 0 \end{cases}$

Proof – left as an exercise in register machine programming!

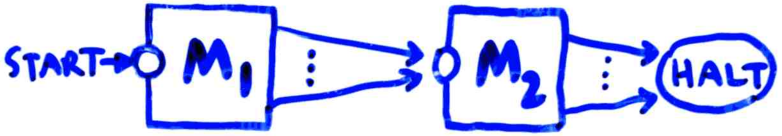
One can solve these kind of problems in a more systematic way by compiling algorithms for the functions written using higher-level control constructs into register machine language.

First note that there is no loss of generality (i.e. the class of computable functions remains unchanged) if we work with register machine programs with exactly one HALT instruction (because...). So graphs of programs look schematically like

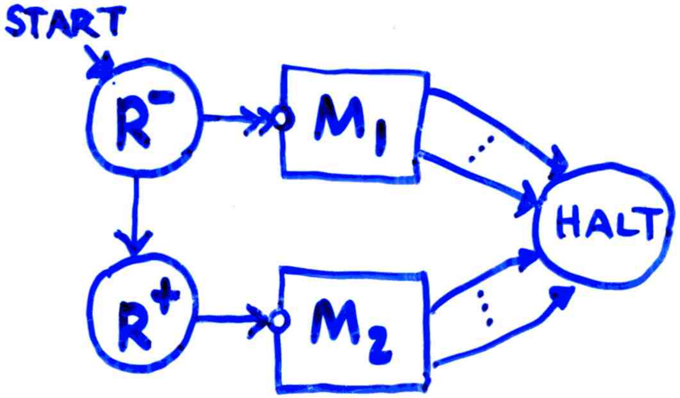


Then we have...

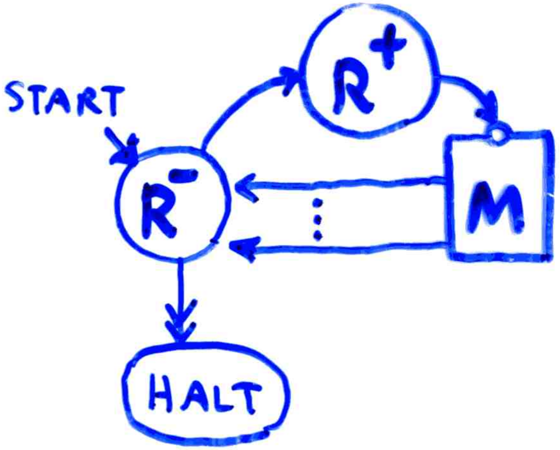
Sequential composition $M_1 ; M_2$



IF $R = 0$ THEN M_1 ELSE M_2



WHILE $R \neq 0$ DO M



A Universal Register Machine.

Part I :

Coding register machines as numbers

A key part of the Turing / Church solution of Hilbert's Entscheidungsproblem was to exploit the idea that

> (formal descriptions of) algorithms can be the data on which algorithms act.

We are using register machines as the formal description of the informal notion of "algorithm". Since the data that register machines manipulate are numbers, to develop the above idea we have to [have an algorithm to] *code* register machines as numbers.*

To do that we need to be able to code $\left\{\begin{array}{l}\text{pairs of numbers}\\ \text{finite lists of numbers}\end{array}\right.$

as numbers. There are many ways of doing that: we fix upon one convenient way ...

---

* such codings are often called *Gödel numberings*, after Gödel's original use of the idea: his coding of arithmetic formulas as numbers was a key part of his proof of the famous Incompleteness Theorem.

Coding pairs of numbers as numbers

For $x, y \in \mathbb{N}$ define

$$\langle\!\langle x, y \rangle\!\rangle \overset{\text{def}}{=} 2^x \cdot (2y+1)$$

$$\langle x, y \rangle \overset{\text{def}}{=} 2^x \cdot (2y+1) - 1$$

Thus

$$\boxed{\langle\!\langle x, y \rangle\!\rangle \text{ in binary}} = \boxed{y \text{ in binary}}\boxed{1}\overbrace{\boxed{0 \cdots 0}}^{x\ 0\text{'s}}$$

$$\boxed{\langle x, y \rangle \text{ in binary}} = \boxed{y \text{ in binary}}\boxed{0}\underbrace{\boxed{1 \cdots 1}}_{x\ 1\text{'s}}$$

Hence

- $\langle\!\langle -, - \rangle\!\rangle$ and $\langle -, - \rangle$ both determine <u>injective</u> functions from $\mathbb{N} \times \mathbb{N}$ to $\mathbb{N}$, i.e.

  $\langle\!\langle x_1, y_1 \rangle\!\rangle = \langle\!\langle x_2, y_2 \rangle\!\rangle \Rightarrow x_1 = x_2 \ \& \ y_1 = y_2$

  $\langle x_1, y_1 \rangle = \langle x_2, y_2 \rangle \Rightarrow x_1 = x_2 \ \& \ y_1 = y_2$

- $\langle\!\langle -, - \rangle\!\rangle$ is a <u>surjective</u> function from $\mathbb{N} \times \mathbb{N}$ to $\{z \in \mathbb{N} \mid z \neq 0\}$, i.e. for all $z \neq 0$ there are $x, y \in \mathbb{N}$ with $\langle\!\langle x, y \rangle\!\rangle = z$.

- $\langle -, - \rangle$ is a surjective function from $\mathbb{N} \times \mathbb{N}$ to $\mathbb{N}$ i.e. for all z there are $x, y \in \mathbb{N}$ with $\langle x, y \rangle = z$.

and hence $\langle -, - \rangle$ is a <u>bijection</u> (a.k.a. one-to-one correspondence) between $\mathbb{N} \times \mathbb{N}$ and $\mathbb{N}$.

(and $\langle\!\langle -, - \rangle\!\rangle$ a bijection between $\mathbb{N} \times \mathbb{N}$ and $\{z \mid z \neq 0\}$).

NOTATION for lists (of numbers)

$\mathbb{N}^* \overset{\text{def}}{=}$ set of finite lists of natural numbers

$$= \underset{\substack{\uparrow \\ \text{unique list} \\ \text{of length 0}}}{\{nil\}} \cup \underset{\substack{\uparrow \\ \text{lists} \\ \text{of length 1}}}{\mathbb{N}} \cup \mathbb{N}^2 \cup \underset{\cdots}{\cdots} \cup \underset{\substack{\uparrow \\ \text{lists } (x_1,\dots,x_n) \\ \text{of length } n}}{\mathbb{N}^n} \cup \cdots$$

$cons \in Fun(\mathbb{N} \times \mathbb{N}^*, \mathbb{N}^*)$

$head \in Pfn(\mathbb{N}^*, \mathbb{N})$

$tail \in Fun(\mathbb{N}^*, \mathbb{N}^*)$

cons is a bijection from $\mathbb{N} \times \mathbb{N}^*$ to $\{\ell \in \mathbb{N}^* \mid \ell \neq nil\}$

$head(cons(x,\ell)) = x$	$head(nil)\uparrow$
$tail(cons(x,\ell)) = \ell$	$tail(nil) = nil$
$\ell = cons(head(\ell), tail(\ell))$ if $\ell \neq nil$	

Every list can be built up from nil by repeated cons's :

$$(x_1, \dots, x_n) = cons(x_1, cons(x_2, \dots, cons(x_n, nil) \cdots))$$

Coding lists $(x_1,\dots,x_n)\in\mathbb{N}^*$ as numbers $[x_1,\dots,x_n]\in\mathbb{N}$

Define $[x_1,\dots,x_n]\in\mathbb{N}$ by induction on the length of the list $(x_1,\dots,x_n)\in\mathbb{N}^*$:

CASE $n=0$: $[\mathrm{nil}] \overset{\mathrm{def}}{=} 0$

INDUCTION STEP : $[\mathrm{cons}(x,\ell)] \overset{\mathrm{def}}{=} \langle x,[\ell]\rangle = 2^x(2[\ell]+1)$

Thus in general $[x_1,\dots,x_n] = \langle x_1,\langle x_2,\dots\langle x_n,0\rangle\cdots\rangle\rangle$

From the definition of $\langle -,-\rangle$ we get :

$$\boxed{[x_1,\dots,x_n] \text{ in binary}} = \boxed{1\,\Big|\,\underbrace{0\cdots0}_{x_n\ 0\text{'s}}\,\Big|\,1\,\Big|\,\underbrace{0\cdots0}_{x_{n-1}\ 0\text{'s}}\,\Big|\ \cdots\ \Big|\,1\,\Big|\,\underbrace{0\cdots0}_{x_1\ 0\text{'s}}}$$

number of 1's in $=$ length of list $(x_1,\dots,x_n)$

Hence $\ell\mapsto[\ell]$ determines a bijection from $\mathbb{N}^*$ to $\mathbb{N}$

Examples

$$[3] = [\mathrm{cons}(3, \mathrm{nil})] = \langle 3, 0 \rangle = 2^3(2 \cdot 0 + 1) = 8 \text{ decimal}$$

$$= 1\underbrace{000}_{3} \text{ binary}$$

$$[1,3] = \langle 1, [3] \rangle = \langle 1, 8 \rangle = 2^1(2 \cdot 8 + 1) = 34 \text{ decimal}$$

$$= 1\underbrace{000}_{3}1\underbrace{0}_{1} \text{ binary}$$

$$[2,1,3] = \langle 2, [1,3] \rangle = \langle 2, 34 \rangle = 2^2(2 \cdot 34 + 1) = 276 \text{ decimal}$$

$$= 1\underbrace{000}_{3}1\underbrace{0}_{1}1\underbrace{00}_{2} \text{ binary}$$

(**NB** reversal of order of list when reading left-to-right in binary representation.)

In order to code register machine programs as numbers, without loss of generality (i.e. without affecting which partial functions can be computed) we can assume:

- the registers of a machine with n+1 registers are always called R0, …, Rn
- the labels occurring in a register machine program are called L0, L1, L2, …, and the $(i+1)^{th}$ instruction in the program listing is labelled Li.

Such codings are often called Gödel numberings

Coding register machine programs Prog numbers $\ulcorner Prog \urcorner \in \mathbb{N}$

If Prog is

$$\boxed{\begin{array}{l} L0 : body_0 \\ L1 : body_1 \\ \quad\vdots \\ Lm : body_m \end{array}}$$

then $\ulcorner Prog \urcorner \overset{def}{=} [\, code(body_0), \ldots, code(body_m) \,]$

where

$$\begin{cases} code(Ri^+ \to Lj) \overset{def}{=} \langle\!\langle 2i, j \rangle\!\rangle \\ code(Ri^- \to Lj, Lk) \overset{def}{=} \langle\!\langle 2i+1, \langle j, k \rangle \rangle\!\rangle \\ \\ code(HALT) \overset{def}{=} 0 \end{cases}$$

Any $x \in \mathbb{N}$ decodes uniquely as an instruction :

if $x = 0$ then the instruction is HALT

else decode x as a pair $x = \langle\!\langle y, z \rangle\!\rangle$ and

if y is even then instruction is

$R_i^+ \to L_j$ where $i = y/2$ & $j = z$

else (y is odd and) decode z as a pair $z = \langle u, v \rangle$ and then the instruction is

$R_i^- \to L_j, L_k$ where $i = (y-1)/2$, $j = u$ & $k = v$.

Hence any $e \in \mathbb{N}$ decodes uniquely as a program Prog_e, called the (register machine) **program with index e** :

First decode e as a list $e = [x_1, \ldots, x_n]$ and then decode each x_i as an instruction, as above.

NB (1) The program resulting from this decoding process may well have jumps to labels greater than the length of the list of instructions, i.e. the associated register machine may well be capable of halting erroneously – but no matter.

(2) In case $e = 0 = [\mathrm{nil}]$ we get an empty list of instructions, which by convention we regard as a machine that does nothing.

Example : decode 666 as a program (for the register machine from hell !)

decimal 666 = binary 1010011010

$= [1, 1, 0, 2, 1]$

Now

0 is code for instruction HALT

$1 = \langle\!\langle 0, 0 \rangle\!\rangle$ is code for instruction $R0^+ \to L0$

$2 = \langle\!\langle 1, 0 \rangle\!\rangle = \langle\!\langle 1, \langle 0, 0 \rangle \rangle\!\rangle$ is code for $R0^- \to L0, L0$

So 666 decodes to the program

$L0 : R0^+ \to L0$
$L1 : R0^+ \to L0$
$L2 : \mathrm{HALT}$
$L3 : R0^- \to L0, L0$
$L4 : R0^+ \to L0$

(which never halts).

A Universal Register Machine.

Part II:

Description of the machine.

High-level description of a universal register machine, U:

- U has registers P(rogram), A(rgument), ...
- Loading P with value e, A with value a and all other registers with 0, then U acts as follows:
  - — decode e as a program: $e = \ulcorner Prog_e \urcorner$
  - — decode a as a list of register values: $a = [a_1, \ldots, a_n]$
  - — carry out the computation of the register machine program $Prog_e$ starting with registers $R0, R1, \ldots, Rn$ set to $0, a_1, a_2, \ldots, a_n$ (and any other registers occurring in $Prog_e$ set to 0).

Overall structure of $\mathcal{U}$'s program :

1 copy P to T, copy PCth item of list in T to N (HALTing if PC > length of list) and goto 2

2 if $N = 0 \; (= code(HALT))$ then HALT, else decode N as $\langle\!\langle y, z \rangle\!\rangle$, assign y to C & z to N, and goto 3

{At this point
either $C = 2i$ is even & current instruction is $R_i^+ \to L_j$ where $j = N$
or $C = 2i+1$ " odd " " " " $R_i^- \to L_j, L_k$ " $\langle j, k \rangle = N$ }

3 remove register values from list in A up to the one required* {the ith}, putting it in R & saving preceding values as a list in S, then goto 4

4 execute current instruction on value in R, update PC {to j or k as above}, restore register values from R and S to A, and goto 1

* see Note on p 42

The registers of $\mathcal{U}$ and the rôle they play in its program:

P - holds the code of the register machine to be simulated
A - holds current contents of registers of the register machine being simulated
PC - program counter - holds the number of the current instruction (counting from 0)

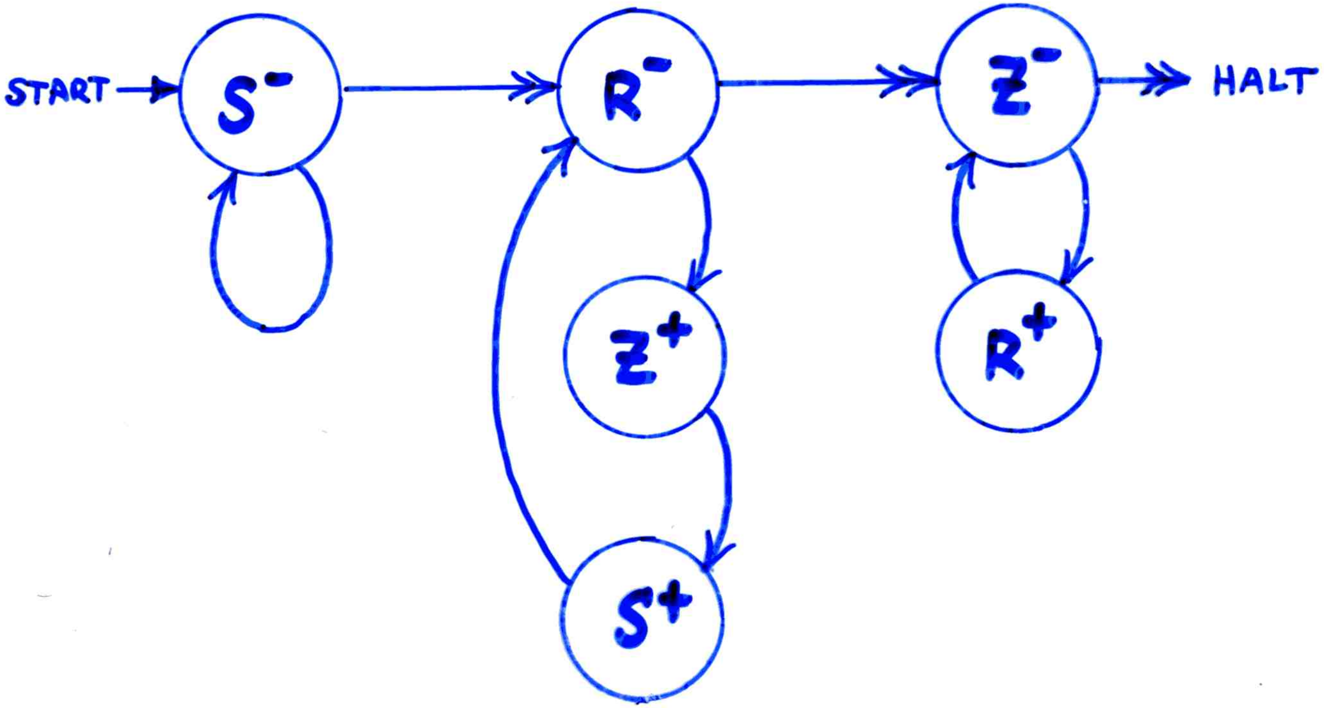
N - holds the code of the current instruction
C - indicates the type of the current instruction
R - holds the contents of simulated machine's register that is to be incremented/decremented by current instruction (if not a HALT instruction)
T - holds a working copy of the program code
S, Z - auxiliary registers for intermediate computations

Note

At step 3 it may be that $i >$ length of the list in A, i.e. that current instruction wants to increment/decrement a register in the simulated machine that was not assigned a value by the initial value of A or has not been mentioned so far in the interpreted program. By assumption, such a register has value 0. So in this case, say $A = [a_1, \dots, a_n]$ with $i > n$, at step 3 A will be set to $0 (= [\mathrm{nil}])$, R set to 0, and S set to $[\underbrace{0, \dots, 0}_{\substack{i-n-1 \\ 0\text{'s}}}, a_n, \dots, a_1]$. Then when the register values are restored at step 4, A will hold $[a_1, \dots, a_n, \underbrace{0, \dots, 0}_{\substack{i-n-1 \\ 0\text{'s}}}, r]$ where r is the value in R after executing the current instruction.

The detailed construction of U's program depends on the fact that various procedures for manipulating (codes of) lists of numbers are register machine programmable ...

The program to carry out $S := R$ can be implemented by



Precondition:

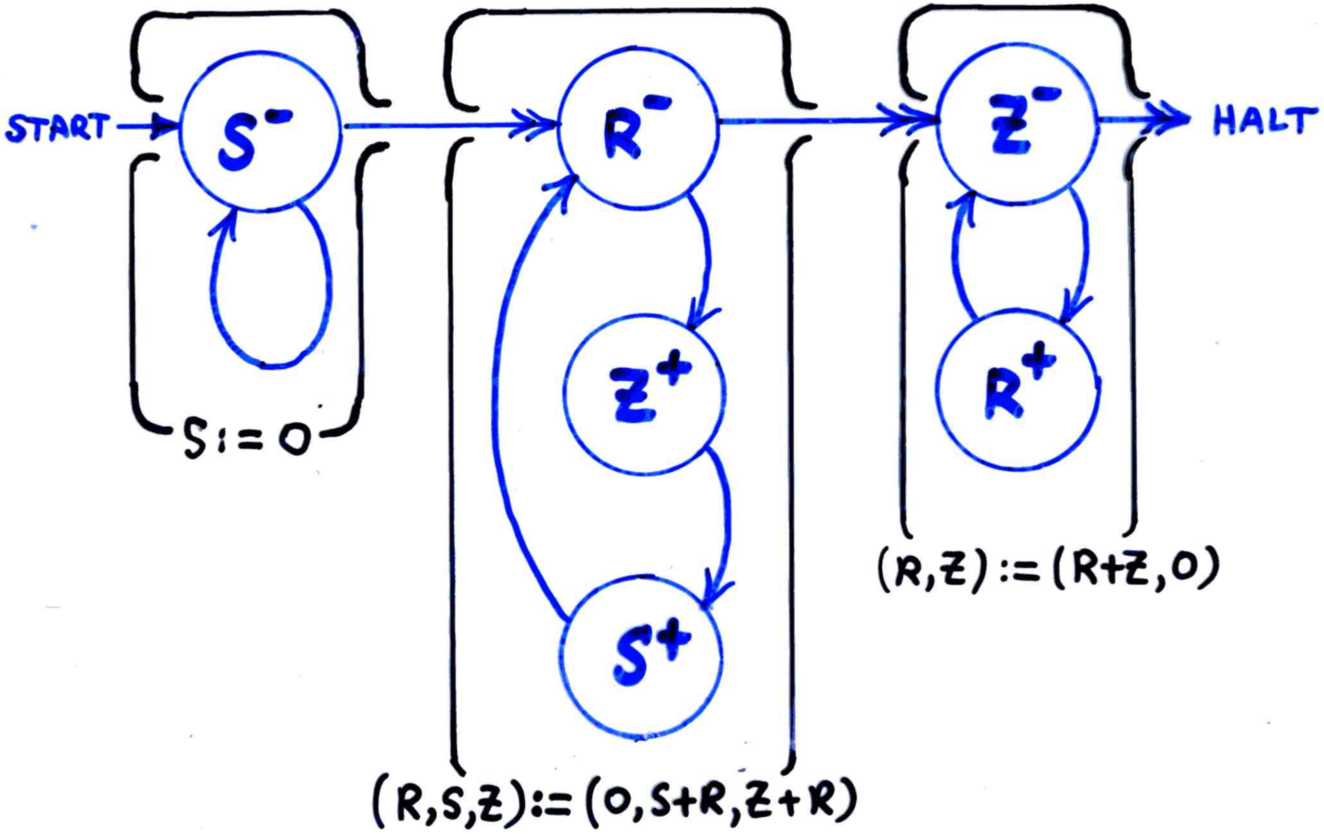
$R = x$
$S = y$
$Z = 0$

Postcondition:

$R = x$
$S = x$
$Z = 0$

The program 

to carry out $S := R$ can be implemented by



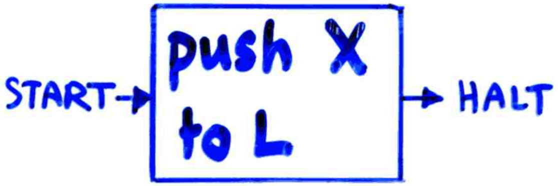
Precondition :

$R = x$
$S = y$
$Z = 0$

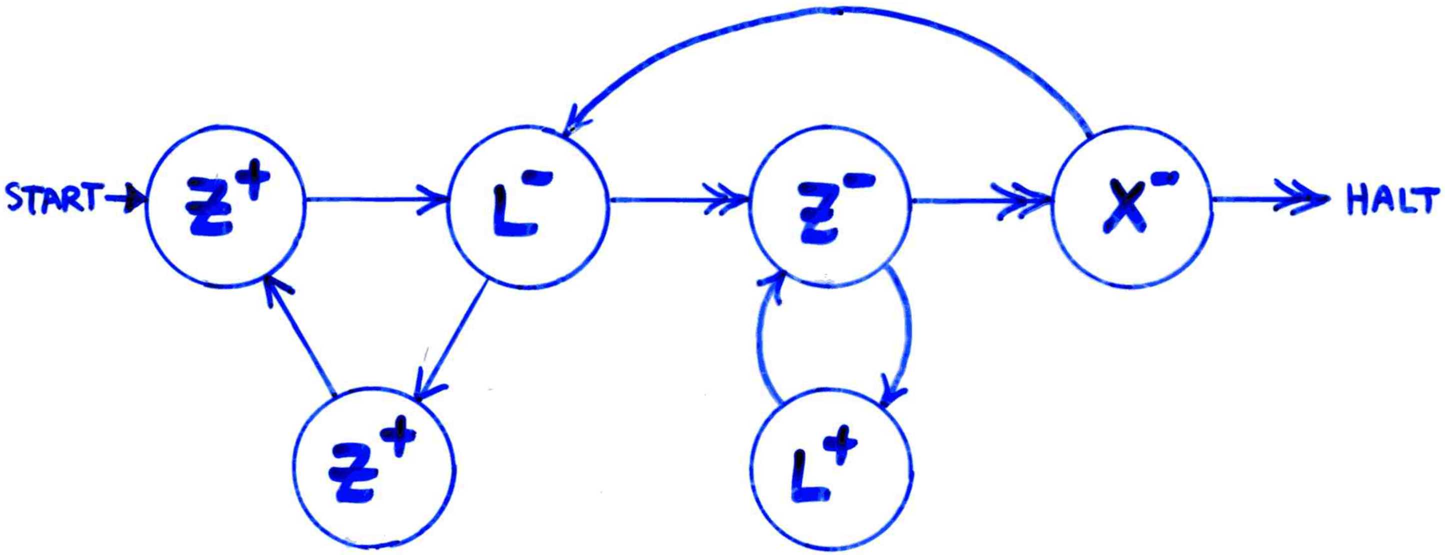
Postcondition :

$R = x$
$S = x$
$Z = 0$

The program



to carry out $(X,L):=(0,\mathrm{cons}(X,L))$ can be implemented by



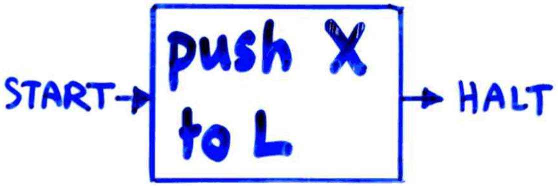
Precondition:

$X = x$
$L = \ell$
$Z = 0$

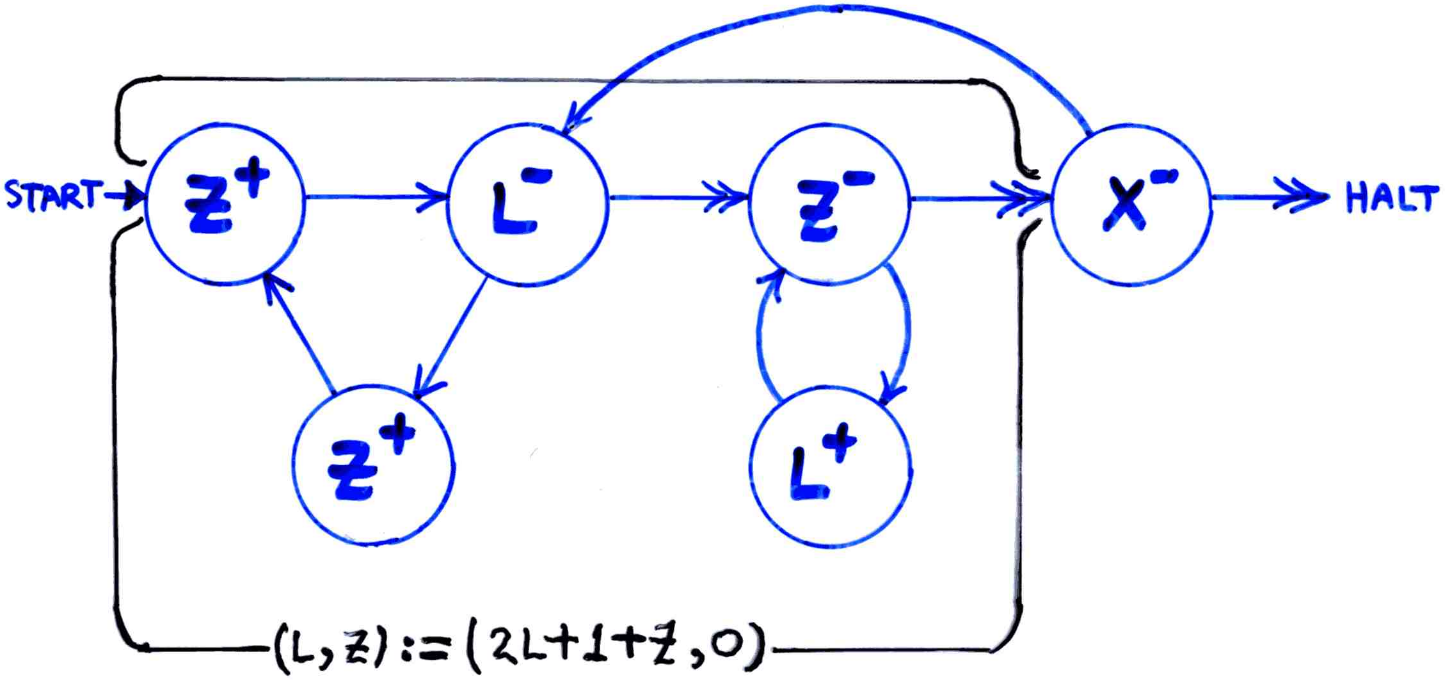
Postcondition:

$X = 0$
$L = \langle x, \ell \rangle = 2^x(2\ell+1)$
$Z = 0$

The program



to carry out $(X,L) := (0, \mathrm{cons}(X,L))$ can be implemented by



Precondition:

$X = x$

$L = \ell$

$Z = 0$

Postcondition:

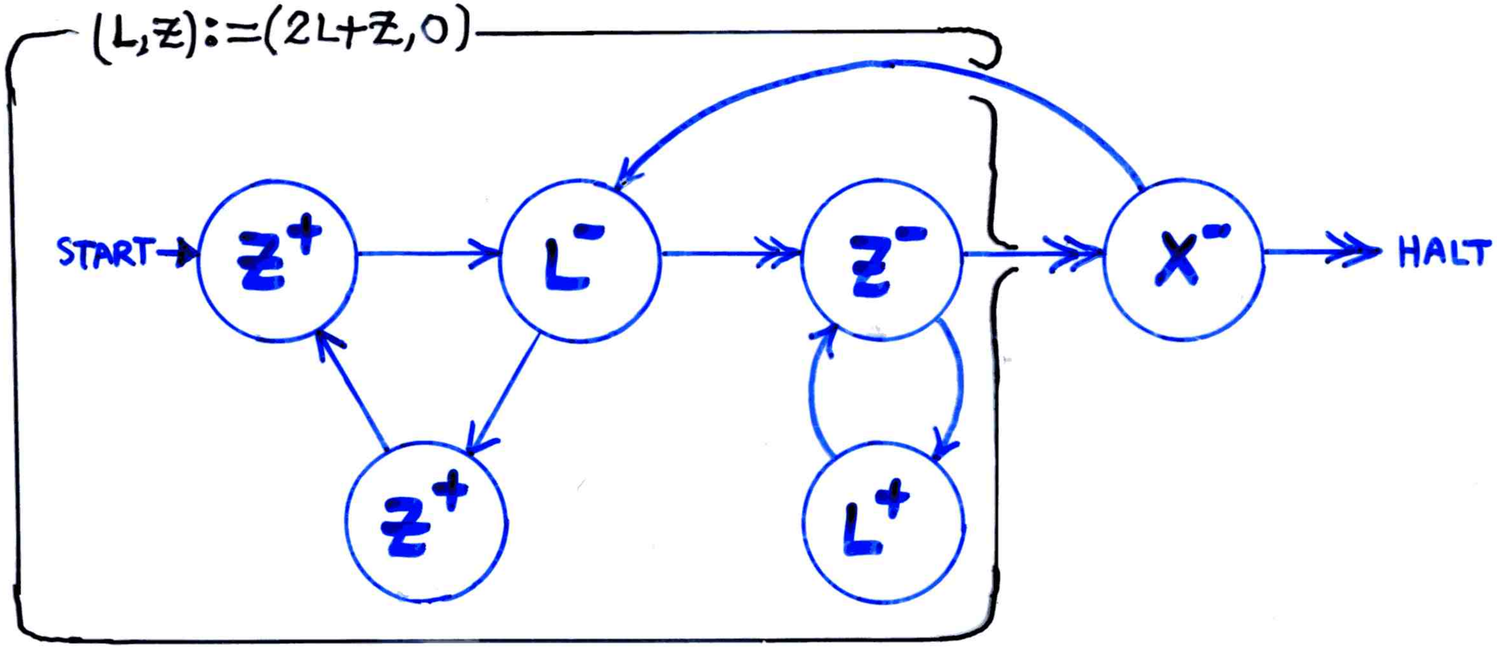
$X = 0$

$L = \langle x, \ell \rangle = 2^x(2\ell+1)$

$Z = 0$

The program

to carry out $(X,L):=(0,\text{cons}(X,L))$ can be implemented by



Precondition:

$X = x$

$L = \ell$

$Z = 0$

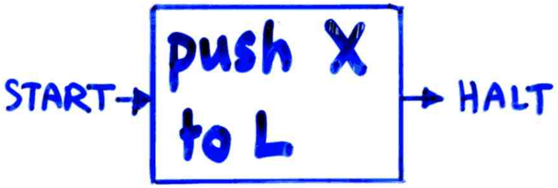
Postcondition:

$X = 0$

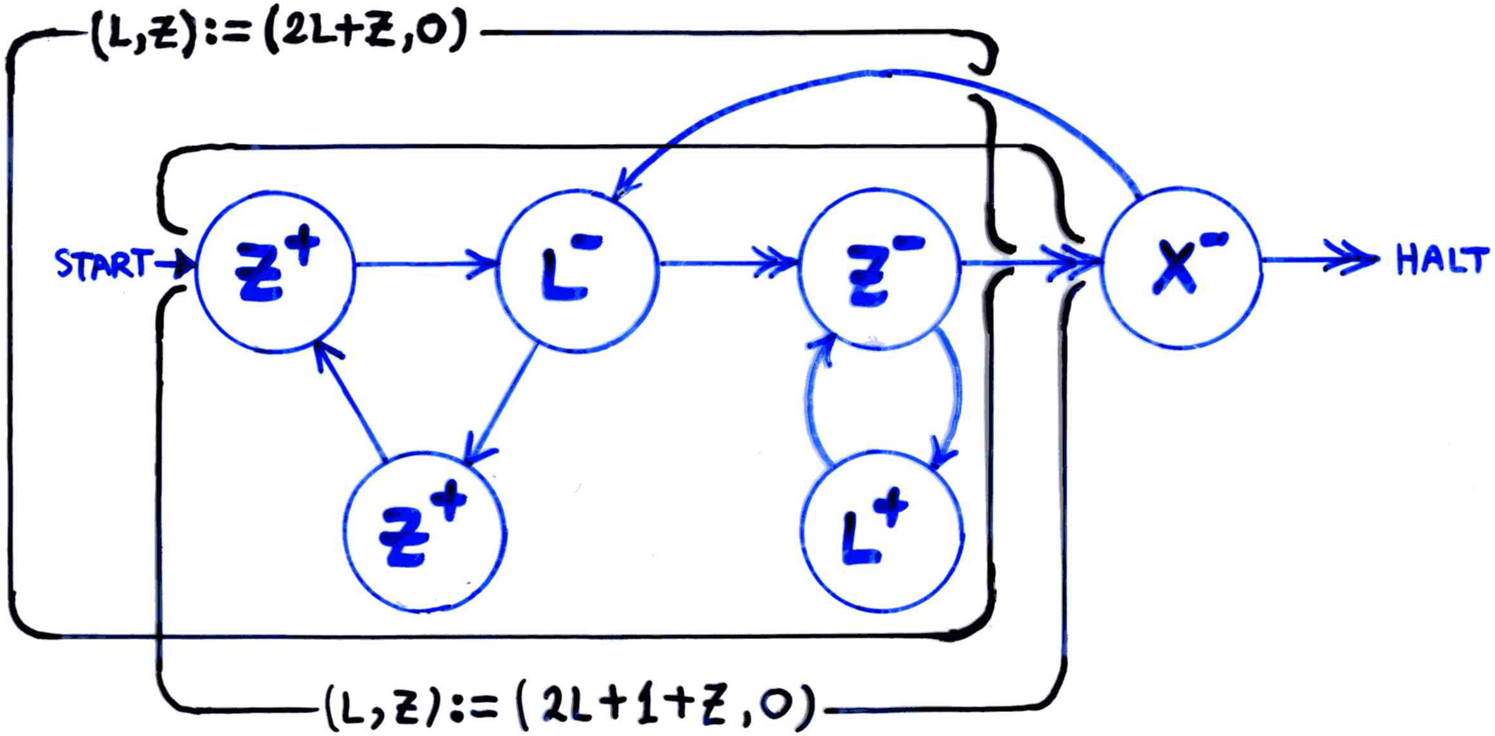
$L = \langle x, \ell \rangle = 2^x(2\ell+1)$

$Z = 0$

The program



to carry out $(X,L):=(0,\mathrm{cons}(X,L))$ can be implemented by



Precondition:

$X = x$

$L = \ell$

$Z = 0$

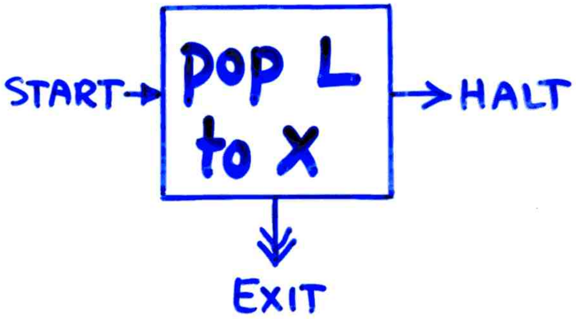
Postcondition:

$X = 0$

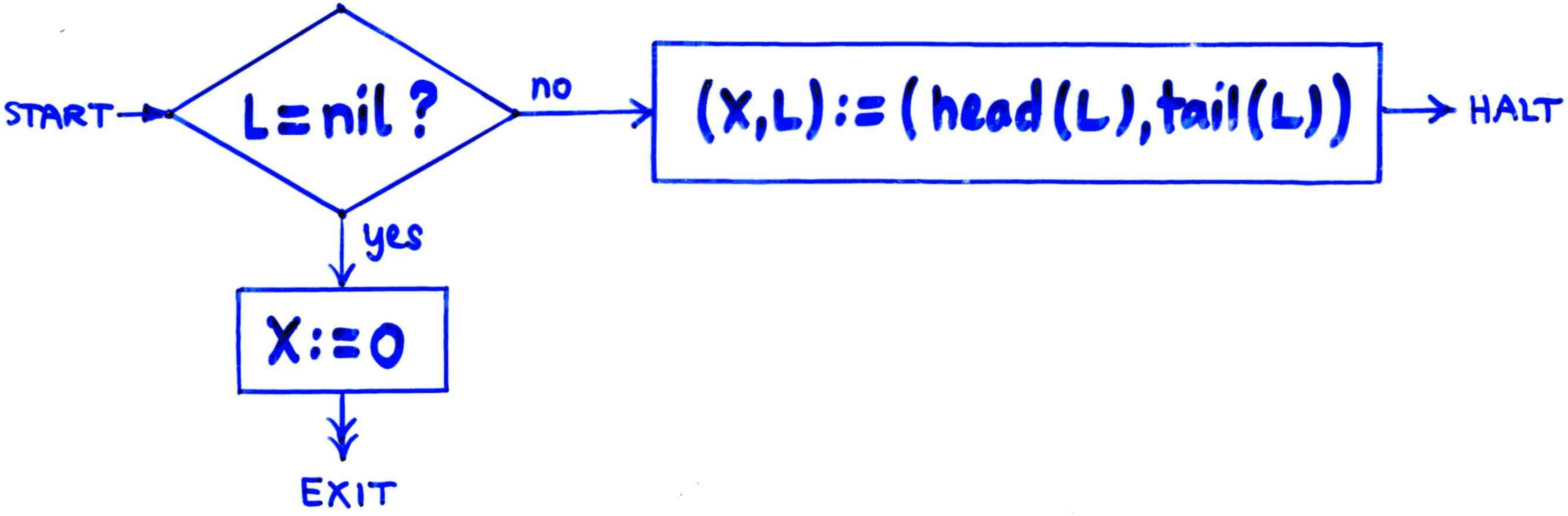
$L = \langle x,\ell \rangle = 2^{x}(2\ell+1)$

$Z = 0$

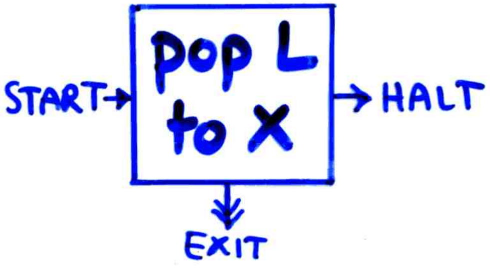
The program

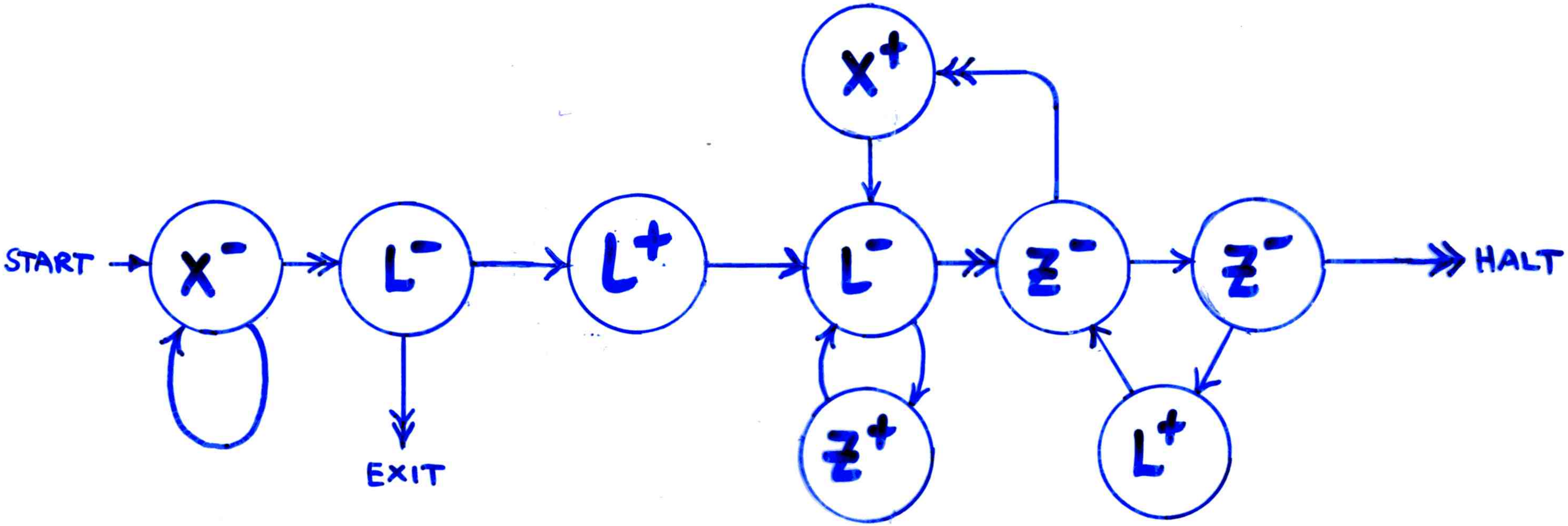


Specification :

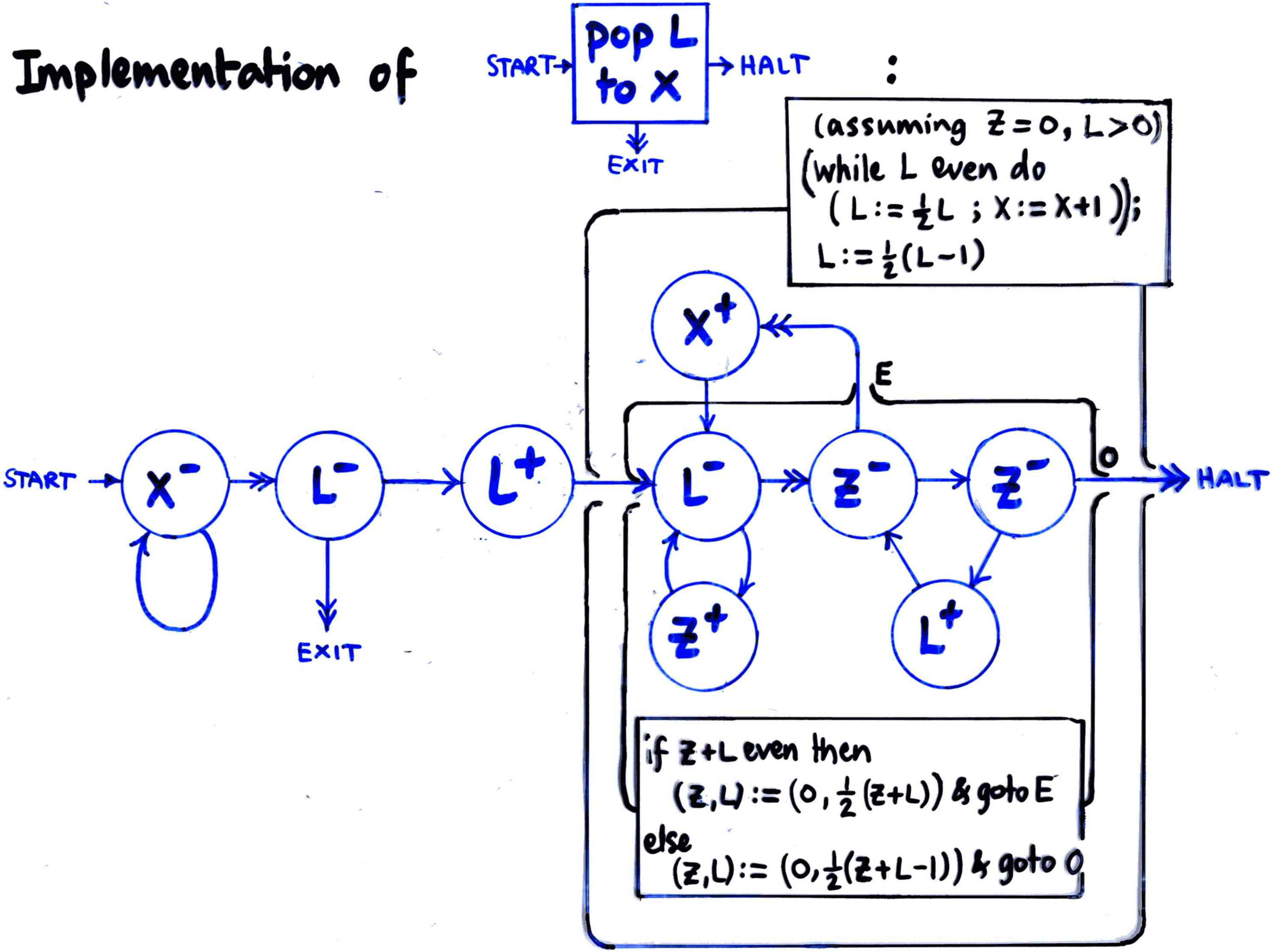


" if $L = 0$ then assign 0 to X and goto EXIT,
else $L = \langle x, \ell \rangle$ say, assign x to X and ℓ to L,
and goto HALT "

Implementation of  :



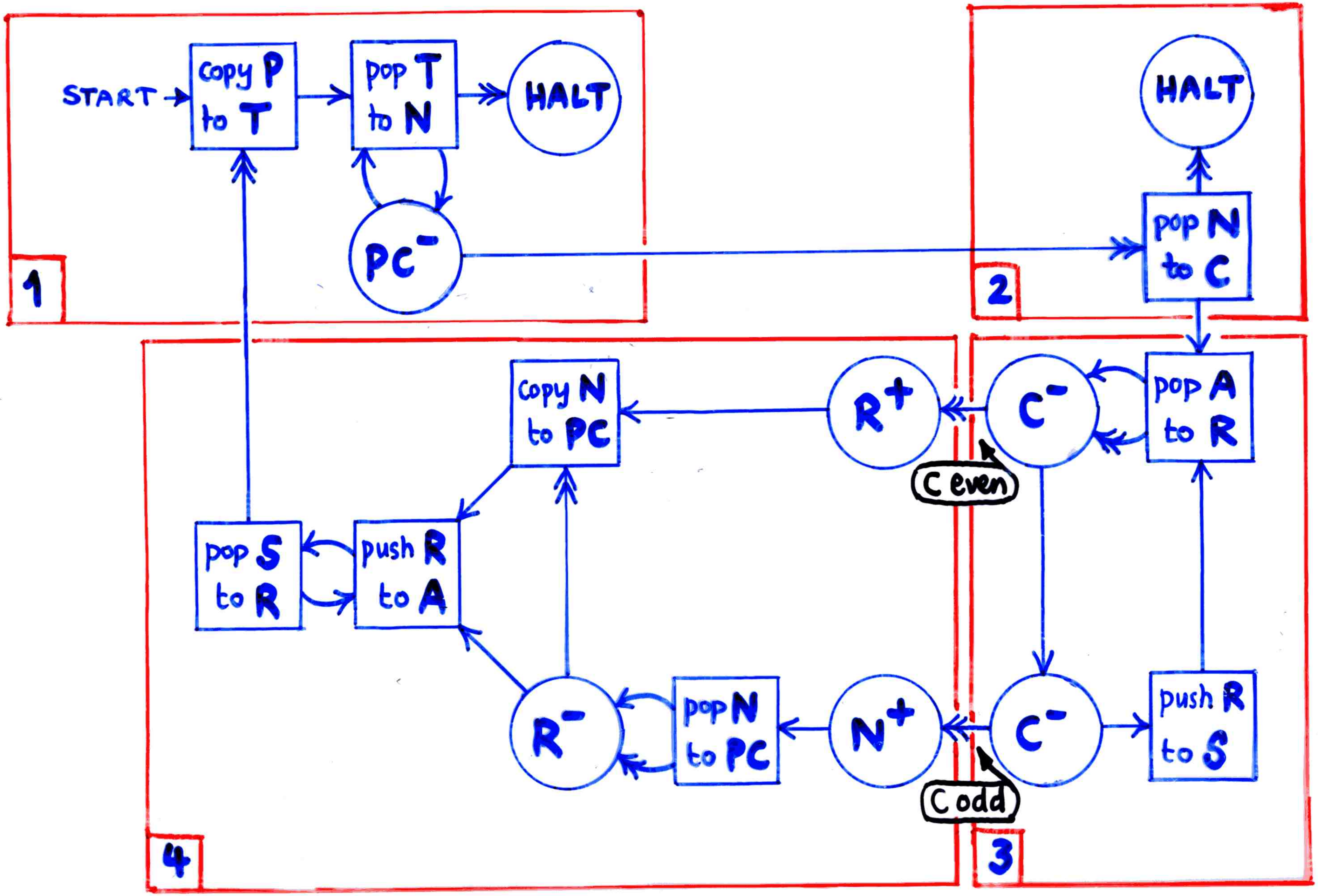
Implementation of pop L to X :



```
(assuming Z = 0, L > 0)
(while L even do
  ( L := ½L ; X := X+1 ));
L := ½(L−1)
```

```
if Z+L even then
  (Z,L) := (0, ½(Z+L)) & goto E
else
  (Z,L) := (0, ½(Z+L−1)) & goto 0
```

The program for $\mathcal{U}$:



The Halting Problem

DEFINITION : a register machine H decides the Halting Problem if, loading R1 with e and R2 with $[a_1, \dots, a_n]$ (and all other registers with 0), the computation of H halts with R0 containing either 0 or 1 ; moreover R0 contains 1 when H halts if & only if

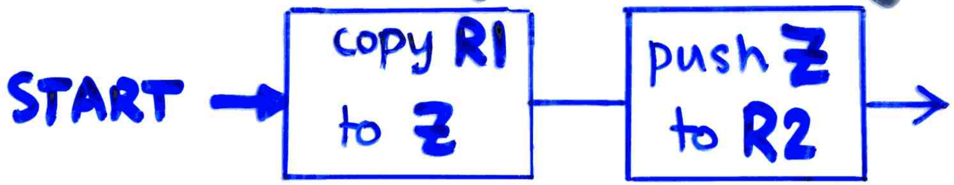
the computation of the register machine program $Prog_e$ started with registers $R1, \dots, Rn$ set to $a_1, \dots, a_n$ (and all other registers set to 0) does halt.

THEOREM : no such register machine H can exist.

Proof :- suppose such an H exists and derive a contradiction ...

Let H′ be obtained from H by replacing

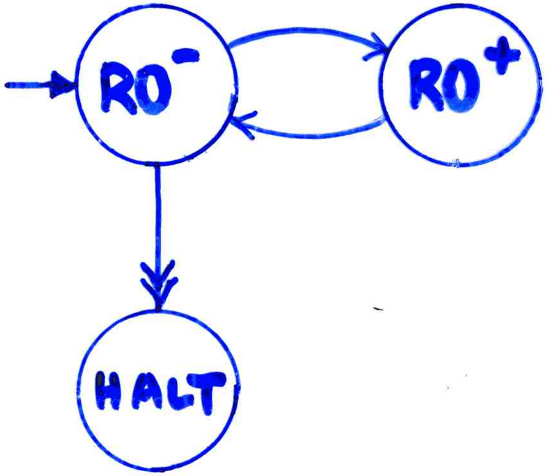
START → by



(where Z is a register not mentioned in H's program)

Let C be obtained from H′ by replacing each HALT (& each jump to a label with no instruction)

by



Let $c \in \mathbb{N}$ be the index of C's program.

C started with $R1=c$ eventually halts

iff

H′ started with $R1=c$ halts with $R0=0$

iff

H started with $R1=c$ & $R2=[c]$ halts with $R0=0$

iff

Prog_c started with $R1=c$ does not halt

iff

C started with $R1=c$ does not halt

CONTRADICTION !

(to the assumption that such an H exists)

Recall:

DEFINITION:

$f \in Pfn(\mathbb{N}^n, \mathbb{N})$ is (register machine) computable if & only if there is a register machine M with at least $n+1$ registers, $R0, R1, R2, \ldots, Rn$ say, (and maybe some other registers as well) with the property that for all $(x_1, \ldots, x_n) \in \mathbb{N}^n$ and all $y \in \mathbb{N}$

$f(x_1, \ldots, x_n) = y$ if & only if the computation of M starting with $R1 = x_1, \ldots, Rn = x_n$, and all other registers $= 0$, halts with $R0 = y$.

Enumerating computable functions

For each $e \in \mathbb{N}$ let $\varphi_e \in \mathrm{Pfn}(\mathbb{N}, \mathbb{N})$ be the partial function computed by Prog_e, i.e.

$\varphi_e(x) = y \overset{\text{def}}{\Longleftrightarrow}$ the computation of Prog_e started with $R1 = x$ (and all other registers zeroed) halts with $R0 = y$

Thus:

the function $e \mapsto \varphi_e$ maps $\mathbb{N}$ onto the collection of all computable partial functions from $\mathbb{N}$ to $\mathbb{N}$.

Not all partial functions are computable

Define $f \in \mathrm{Pfn}(\mathbb{N},\mathbb{N})$ by :

$$f(e) \overset{\text{def}}{=} \begin{cases} 0 & \text{if } \varphi_e(e)\uparrow \\ \text{undefined} & \text{if } \varphi_e(e)\downarrow \end{cases}$$

CLAIM : f is not computable.

PROOF : If f computable, then $f = \varphi_e$ for some e.

Then

- $\varphi_e(e)\uparrow \xRightarrow[f]{\text{def}^{\text{n}}\text{ of}} f(e)=0 \xRightarrow[e]{\text{def}^{\text{n}}\text{ of}} \varphi_e(e)=0 \Rightarrow \varphi_e(e)\downarrow$
- $\varphi_e(e)\downarrow \Longrightarrow f(e)\uparrow \Rightarrow \varphi_e(e)\uparrow$

Contradiction!

(Un)decidable sets of numbers

A subset $S \subseteq \mathbb{N}$ is (register machine) decidable if & only if there is a register machine M with the property: for all $x \in \mathbb{N}$, M started with $R1 = x$ (and other registers zeroed) always halts with $R0$ containing either 0 or 1; moreover $R0 = 1$ when M halts if and only if $x \in S$.

Equivalently: S is decidable if & only if there is some e such that for all $x \in \mathbb{N}$

$$\text{either } (\varphi_e(x) = 0 \ \& \ x \notin S) \text{ or } (\varphi_e(x) = 1 \ \& \ x \in S)$$

S is called undecidable if no such M (or e) exists.

Some examples of undecidable sets of numbers

$$S_1 \overset{\text{def}}{=} \{\langle e, a\rangle \mid \varphi_e(a)\downarrow\}$$

i.e. one-argument version of Halting Problem

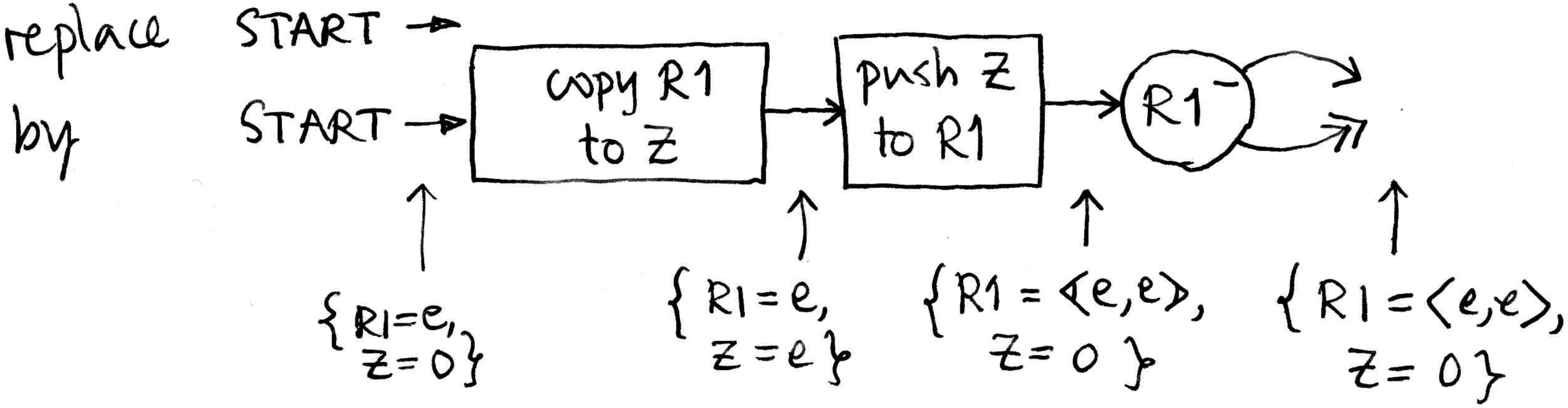
$$S_2 \overset{\text{def}}{=} \{e \mid \varphi_e(0)\downarrow\}$$

i.e. ∄ register machine to decide whether any program halts when supplied with input 0

$$S_3 \overset{\text{def}}{=} \{e \mid \varphi_e \text{ is a total function}\}$$

i.e. ∄ register machine to decide whether any program halts for <u>all</u> input data

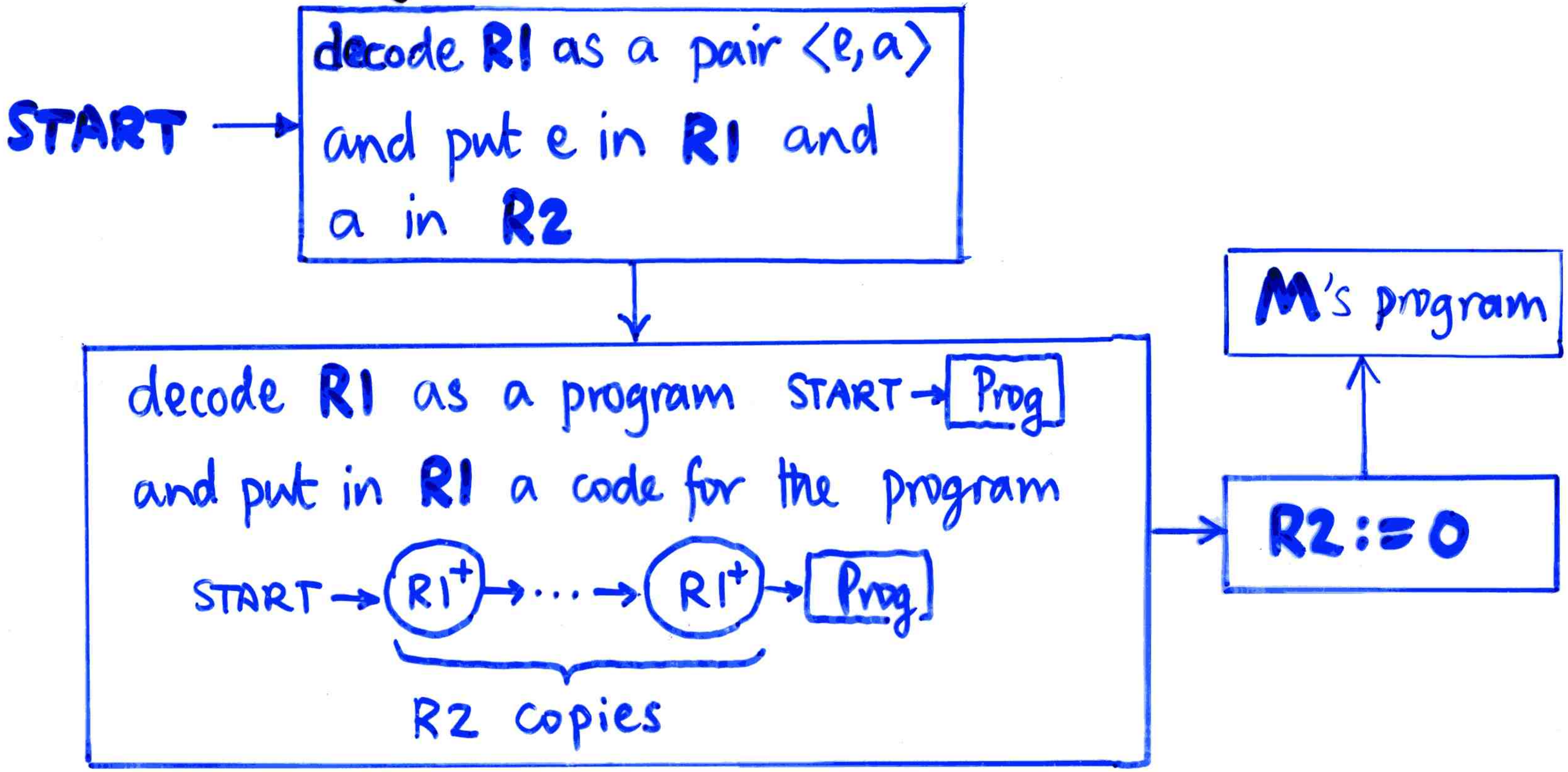
Ex. 1. The proof that S_1 is undecidable is like the proof of the undecidability of the n-argument Halting Problem given above, except that now the modification of H to H′ is:



(the rest of the argument is the same).

Ex. 2. Undecidability of S_2 can be reduced to the undecidability of S_1 :

If M were a register machine for deciding membership of S_2, then the register machine specified by



would decide membership of S_1. So no such M exists.

Remark. We can restate the proof of Ex. 2 in terms of functions: it suffices to show that there is a function $f \in \mathrm{Fun}(\mathbb{N}, \mathbb{N})$ satisfying

- f is computable
- for all $e, a \in \mathbb{N}$ $\quad \varphi_{f(\langle e, a\rangle)}(0) \equiv \varphi_e(a)$

[meaning left hand side ↓ if & only if right hand side ↓ and in that case they are equal (see page 89)]

and hence $\langle e, a\rangle \in S_1 \Leftrightarrow f(\langle e, a\rangle) \in S_2$.

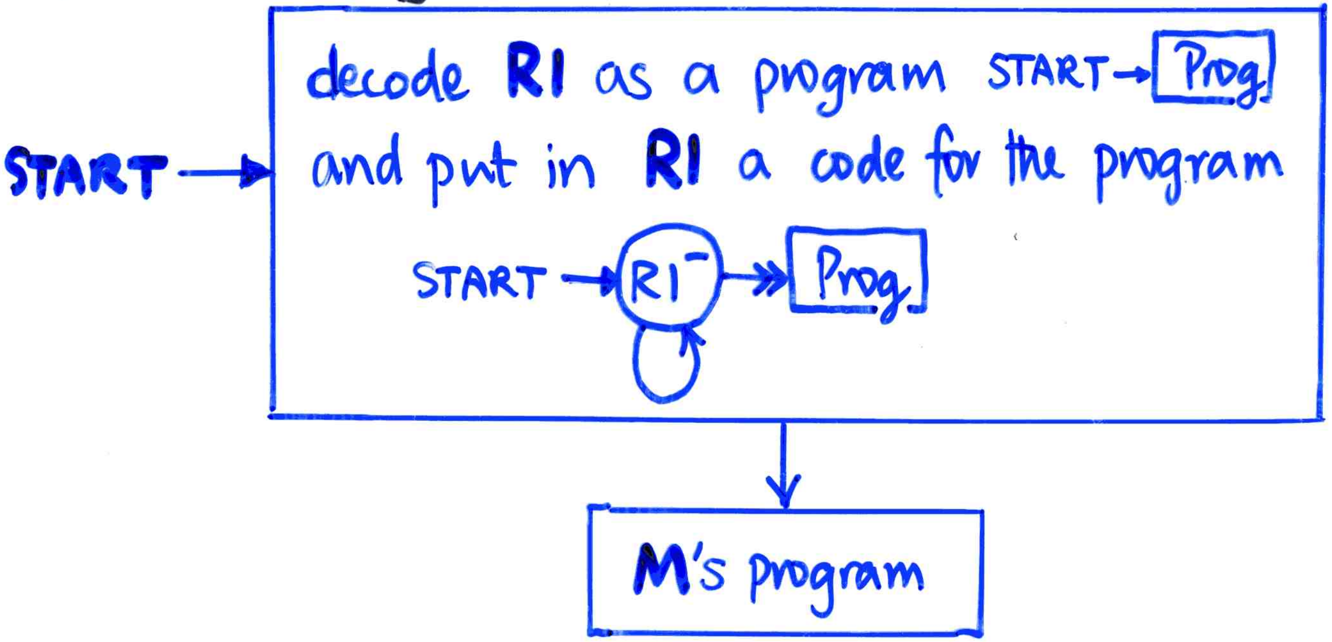
For in general we have for subsets $S_1, S_2 \subseteq \mathbb{N}$

$$\boxed{\begin{array}{l} S_2 \text{ decidable}, f \text{ computable } \& \ \forall x \in \mathbb{N}.\ x \in S_1 \Leftrightarrow f(x) \in S_2 \\ \Rightarrow S_1 \text{ decidable} \end{array}}$$

(why?)

Ex.3. Undecidability of S_3 can be reduced to that of S_2:

If M were a register machine for deciding membership of S_3, then the register machine specified by



decode R1 as a program START→ Prog and put in R1 a code for the program START → $R1^-$ → Prog.

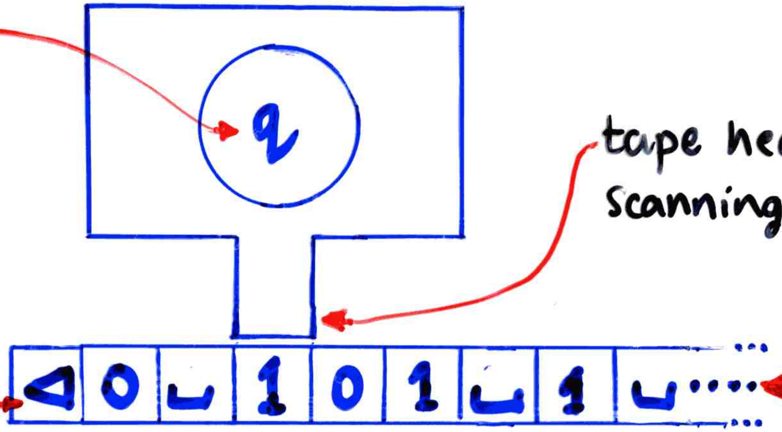
would decide membership of S_2. So no such M exists.

Turing machines
and the
Church-Turing Thesis

Register machine computation takes for granted that we have some concrete representation of the natural numbers and the elementary operations on them of increment, zero test and decrement.

Turing's original model of computation — now called a Turing machine — formalizes the intuitive notion of algorithm in the most concrete terms possible (Turing argued), where even numbers have to be represented explicitly in terms of a fixed, finite alphabet of symbols (eg unary notation, binary notation, etc.) and the elementary operations have to be given explicitly in terms of elementary symbol-manipulating operations ...

Turing machines – informal description



machine is in one of a finite set K of states

tape head is scanning symbol 1

special first symbol

linear tape, unbounded to the right, and divided into cells, each containing a symbol from a finite alphabet Σ, only finitely many of which are not the special blank symbol, ␣.

- machine starts with tape head pointing to ◁
- machine computes in discrete steps, each of which depends only on current state & symbol being scanned by tape head.
- action at each step (if any) is:

  overwrite current tape cell with a symbol,
  move left or right one cell, or stay stationary, and
  change to another state.

DEFINITION: a **Turing machine** consists of:

- a finite set Σ of **tape symbols**, containing distinguished elements $\begin{cases} \sqcup = \textbf{blank symbol} \\ \triangleleft = \textbf{first symbol} \end{cases}$
- a finite set K of **machine states** (disjoint from Σ) containing a special element s = **initial state**
- a function

$$\delta \in \mathrm{Fun}(K \times \Sigma, (K \cup \{acc, rej\}) \times \Sigma \times \{L, R, S\})$$

called the **transition function** of the machine, satisfying: for all q, q', a', D
if $\delta(q, \triangleleft) = (q', a', D)$,
then $a' = \triangleleft$ and $D = R$

(N.B. $acc, rej \notin K$ are special accepting/rejecting states.)

Example of a Turing Machine (p68)

$\Sigma = \{\triangleleft, \sqcup, 0, 1\}$

$K = \{s, q, q'\}$

δ given by :

	$\triangleleft$	$\sqcup$	0	1
s	$(s, \triangleleft, R)$	$(q, \sqcup, R)$	$(rej, 0, S)$	$(rej, 1, S)$
q	$(rej, \triangleleft, R)$	$(q', 0, L)$	$(q, 1, R)$	$(q, 1, R)$
q'	$(rej, \triangleleft, R)$	$(acc, \sqcup, S)$	$(rej, 0, S)$	$(q', 1, L)$

The transition function δ specifies how the Turing machine should act at each step. If

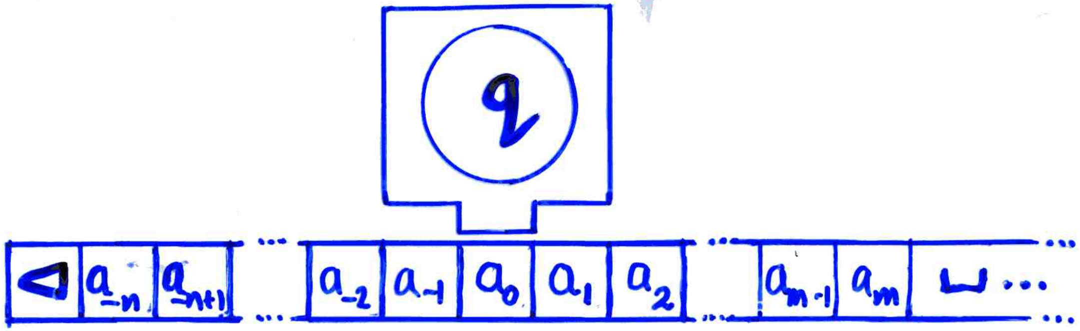
current state $= q \in K$
current tape symbol $= a \in \Sigma$

and $\delta(q,a) = (q', a', D)$
then:

- if $q' = acc$, the machine has reached accepting state (& stops after overwriting a with a')
- if $q' = rej$, the machine has reached rejecting state (& stops after owriting a with a')
- if $q' \in K$, the machine
  - overwrites a with a',
  - changes to state q', and
  - moves one cell Left if $D = L$
    " " " Right " $D = R$
    stays Stationary " $D = S$.

(N.B. because of the condition on δ in the definition on p63, if $a = \lhd$, then machine never overwrites $\lhd$ with anything else and always moves to the right.)

A Turing machine configuration



can be specified by a triple

$$(q, \ell, r)$$

current state $\in K \cup \{acc, rej\}$

finite, non-nil list of elements of Σ

$\ell = (a_0, a_{-1}, a_{-2}, \dots, a_{-n}, \lhd)$ starting with current symbol, reading right-to-left, ending with $\lhd$

finite (possibly nil) list of elements of Σ

$\ell = (a_1, a_2, \dots, a_m)$ starting with symbol to right of current one, reading left-to-right, ending at some point where there are only blanks to the right.

Transition relation $(q_1, \ell_1, r_1) \rightarrow (q_2, \ell_2, r_2)$ is defined to hold if & only if q_1 ℓ_1 r_1 q_2 ℓ_2 r_2 match one of the following cases:

- $(q, cons(a, \ell), r) \rightarrow (q', \ell, cons(a', r))$
  where $\delta(q, a) = (q', a', L)$
- $(q, cons(a, \ell), nil) \rightarrow (q', cons(\sqcup, cons(a', \ell)), nil)$
  where $\delta(q, a) = (q', a', R)$
- $(q, cons(a, \ell), cons(b, r)) \rightarrow (q', cons(b, cons(a', \ell)), r)$
  where $\delta(q, a) = (q', a', R)$
- $(q, cons(a, \ell), r) \rightarrow (q', cons(a', \ell), r)$
  where $\delta(q, a) = (q', a', S)$

A computation of the Turing machine consists of a finite or infinite sequence of transitions between configurations:

$$(\underset{\substack{\uparrow \\ \text{initial} \\ \text{state}}}{s}, \underset{\substack{\uparrow \\ \text{one-element list} \\ \text{just containing } \triangleleft}}{\triangleleft}, r) \to (q_1, l_1, r_1) \to (q_2, l_2, r_2) \to \cdots$$

The computation <u>does not halt</u> if the sequence is infinite

" " <u>halts</u> " " " " finite, in which case the last configuration is of the form (acc, l, r) or (rej, l, r).

EXAMPLE

Consider the Turing machine with

$\Sigma = \{\lhd, \sqcup, 0, 1\}$, $K = \{s, q, q'\}$

and δ given by:

	$\lhd$	$\sqcup$	0	1
s	$(s, \lhd, R)$	$(q, \sqcup, R)$	$(rej, 0, S)$	$(rej, 1, S)$
q	$(rej, \lhd, R)$	$(q', 0, L)$	$(q, 1, R)$	$(q, 1, R)$
q'	$(rej, \lhd, R)$	$(acc, \sqcup, S)$	$(rej, 0, S)$	$(q', 1, L)$

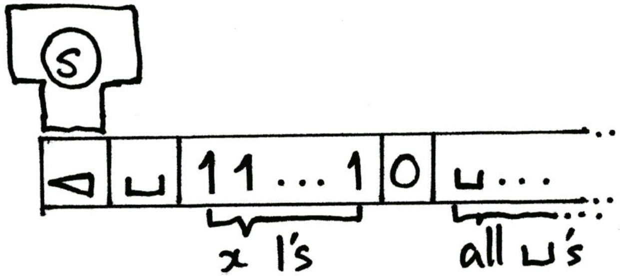
list $(\sqcup, \overbrace{1, \dots, 1}^{x\ 1's}, 0)$

Then the computation starting from configuration $(s, \lhd, \sqcup\bar{x}0)$

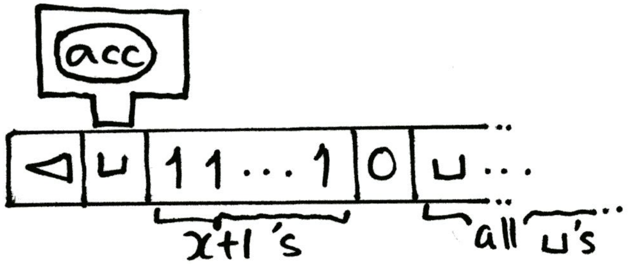
halts in configuration $(acc, \sqcup\lhd, \overline{x+1}0)$.

Example, continued.]

So the starting configuration looks like



the final configuration looks like



and the transitions between are

$$
\begin{aligned}
(s, \triangleleft, \sqcup\bar{x}0) &\to (s, \sqcup\triangleleft, \bar{x}0) \\
&\to (q, 1\sqcup\triangleleft, \overline{x-1}0) \\
&\;\vdots \\
&\to (q, \bar{x}\sqcup\triangleleft, 0) \\
&\to (q, 0\bar{x}\sqcup\triangleleft, \text{nil}) \\
&\to (q, \sqcup\overline{x+1}\sqcup\triangleleft, \text{nil}) \\
&\to (q', \overline{x+1}\sqcup\triangleleft, 0) \\
&\;\vdots \\
&\to (q', \sqcup\triangleleft, \overline{x+1}0) \\
&\to (\text{acc}, \sqcup\triangleleft, \overline{x+1}0)
\end{aligned}
$$

tape head moving right

tape head moving left

PROPOSITION :

The computation of a Turing machine can be implemented on a register machine.

Proof

First, represent tape and state symbols of the Turing machine by numbers, say:

$acc = 0$ $\qquad$ $\sqcup = 0$

$rej = 1$ $\qquad$ $\triangleleft = 1$

$s = 2$

$K = \{2, 3, \ldots, n\}$ $\qquad$ $\Sigma = \{0, 1, \ldots, m\}$

Then code Turing machine configurations (q, ℓ, r) as numbers $[q, [\ell], [r]]$ (using the coding of lists of numbers as numbers that we developed earlier). Since the transition function $\delta \in Fun(K \times \Sigma, (K \cup \{acc, rej\}) \times \Sigma \times \{L, R, S\})$ is a *finite* set (of (argument, result)-pairs), one can construct a register machine program operating on registers

S (for state), T (for tape symbol), D $\left(\text{for direction, with } \begin{cases} L = 0 \\ R = 1 \\ S = 2 \end{cases} \text{ say}\right)$

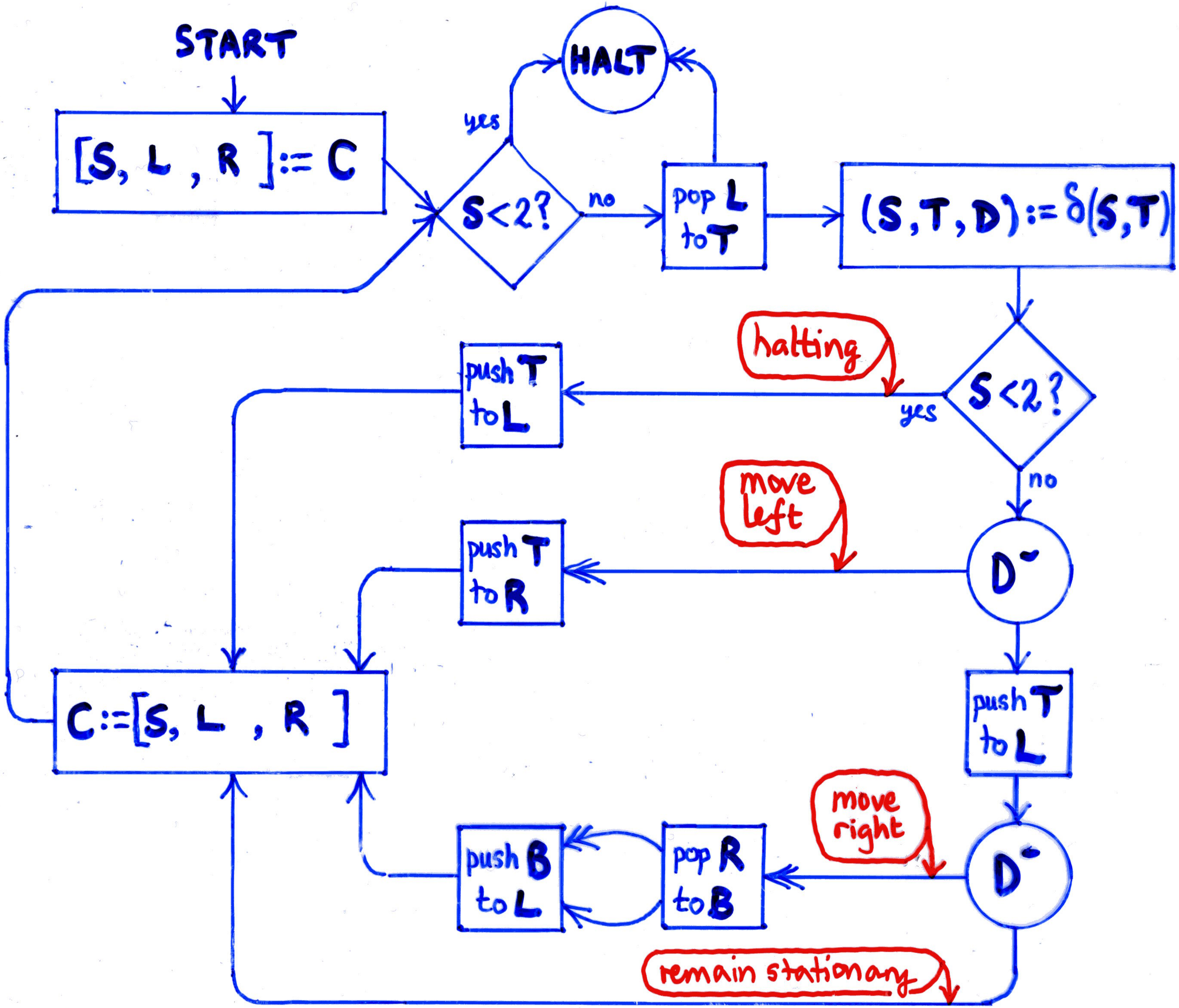
implementing

$$\to \boxed{(S, T, D) := \delta(S, T)} \to$$

Then using registers

C to hold codes of Turing machine configurations
L to hold codes of tape symbol list at & to left of tape head
R " " " " " " to right of " "

the computation of the Turing machine is carried out by the register machine specified on the next page — i.e. starting the register machine with C holding the code of the initial configuration (& all other registers zeroed), the register machine halts if & only if the corresponding Turing machine computation halts, and in that case C holds the code of the final configuration. □


START
[S, L , R]:= C
S<2?
yes
HALT
no
pop L to T
(S,T,D):= δ(S,T)
halting
yes
S<2?
no
push T to L
move left
D˘
push T to R
push T to L
move right
D˘
pop R to B
push B to L
C:=[S, L , R]
remain stationary

Recall:

DEFINITION:

$f \in \mathrm{Pfn}(\mathbb{N}^n, \mathbb{N})$ is (register machine) computable if & only if there is a register machine M with at least $n+1$ registers, $R0, R1, R2, \ldots, Rn$ say, (and maybe some other registers as well) with the property that for all $(x_1, \ldots, x_n) \in \mathbb{N}^n$ and all $y \in \mathbb{N}$

$f(x_1, \ldots, x_n) = y$ if & only if the computation of M starting with $R1 = x_1, \ldots, Rn = x_n$, and all other registers $= 0$, halts with $R0 = y$.

We have seen that Turing machine computation can be implemented by register machines. The converse also holds: the computation of a register machine can be implemented by a Turing machine.

To make sense of this statement, we first have to fix a tape representation of register contents, i.e. a tape representation of finite lists of numbers:

- we will use unary notation for individual numbers:

$$\text{number } x \quad \leftrightarrow \quad \underbrace{11\cdots 1}_{x\ 1\text{'s}}$$

- we will use 0 to mark the beginning and end of a list
- we will use ␣ (the blank symbol) to separate numbers in the list.

Thus we can take the alphabet of tape symbols to be $\Sigma = \{\triangleleft, ␣, 0, 1\}$ and then...

A tape over $\{\lhd, \sqcup, 0, 1\}$ **codes a list of numbers** if & only if

precisely two cells contain 0 and the only cells containing 1 occur between these

Such tapes look like :

we call this cell of such a tape the **initial 0 cell**

$$\lhd \; \sqcup \cdots \sqcup \; 0 \; \underbrace{1 \cdots 1}_{x_1 \text{ 1's}} \; \sqcup \; \underbrace{1 \cdots 1}_{x_2 \text{ 1's}} \; \sqcup \cdots \sqcup \; \underbrace{1 \cdots 1}_{x_n \text{ 1's}} \; 0 \; \underbrace{\sqcup \sqcup \cdots}_{\text{all } \sqcup\text{'s}}$$

and the corresponding list of numbers is :

$$\mathrm{cons}(x_1, \mathrm{cons}(x_2, \ldots, \mathrm{cons}(x_n, \mathrm{nil}) \cdots))$$

i.e. $(x_1, x_2, \ldots, x_n)$

DEFINITION:

$f \in \mathrm{Pfn}(\mathbb{N}^n, \mathbb{N})$ is Turing computable if & only if there is a Turing machine T with the following property:

Starting T from its initial state with tape head on the first symbol $\triangleleft$ of a tape coding $(0, x_1, \ldots, x_n)$, T halts if & only if $f(x_1, \ldots, x_n)\downarrow$, and in that case the final tape codes a list (of length ≥ 1) whose first element is y where $f(x_1, \ldots, x_n) = y$.

THEOREM: A partial function is Turing computable if & only if it is register machine computable.

Proof

Since we can implement any Turing machine by a register machine, it follows that

Turing computable ⇒ register machine computable.

To see that

" " ⇐ " " "

one has to implement the computation of a register machine in terms of a Turing machine operating on a tape coding instantaneous register contents. To do this, one has to see how to carry out the action of each type of register machine instruction on the tape representation of register contents. It should be reasonably clear that this is possible in principle, even if the details (which we omit) are somewhat tedious. □

CHURCH-TURING THESIS:

Every algorithm (in the intuitive sense) can be realized as a Turing machine.

Or, equivalently, every algorithm can be realized as a register machine.

The Church-Turing Thesis is not a statement that can be proved formally – because it refers to the informal notion of "algorithm".

Turing gave a closely argued justification that his machines captured the fundamental elements of the notion of algorithm. Since his time much empirical evidence has accumulated to support the Church-Turing Thesis:

- Several extensions of the notion of Turing machine (and register machine) that have been proposed (e.g. extensions by non-deterministic features or by parallel computation) have all been shown to have equivalent computing power to the original formulation.

- A number of alternative formalizations of the intuitive notion of algorithm (some of which appear quite unconnected with the Turing/register machine formalism) have turned out to determine the same collection of computable functions:

Some approaches to computability

- Church (1936): (untyped) lambda calculus & λ-definable functions.
  [see: CST IB "Foundations of Functional Programming" course.]
- Turing (1936): Turing machines.
- Gödel-Kleene (1936): partial recursive functions
- Post (1943): canonical systems for generating theorems in a formal system.
- Markov (1951): deterministic version of Post's canonical systems.
- Lambek (1961), Minsky (1961) : register machines.
  Shepherdson-Sturgis (1963)

Church's (untyped) λ-calculus

[overview! not examinable]

λ-terms $$M ::= \overset{\text{variables}}{x} \mid \overset{\text{function abstractions}}{\lambda x.M} \mid \underset{\text{function applications}}{M(M)}$$

β-reduction $M \to M'$: smallest relation s.t.

$$\lambda x.M\,(N) \to \overset{\text{substitution}}{M[N/x]}$$

if $M \to M'$ then $N(M) \to N(M')$ & $M(N) \to M'(N)$ & $\lambda x.M \to \lambda x.M'$

---

n^{th} Church numeral : $\ulcorner n \urcorner = \lambda y.\lambda x.\underbrace{y(y(\cdots(y}_{n\ \text{"}y\text{"'s}}(x))\cdots)$

$f \in Pfn(\mathbb{N},\mathbb{N})$ is λ-definable iff ∃ λ-term F so that

$$f(x)=y \iff F(\ulcorner x \urcorner) \to \cdots \to \ulcorner y \urcorner$$

$\boxed{\text{THEOREM : λ-definable = computable}}$

All of the above approaches give rise to the same collection of partial functions from numbers to numbers.
The same is true for any "general purpose" programming language (indeed, one usually takes as a definition of "general purpose"* that the language can code any computable partial function).

We will look at one of the above, alternative approaches in some detail – namely the Gödel-Kleene characterization of computable functions as "partial recursive" functions ...

---

* or "Turing powerful"

Primitive recursive functions

Aim to give a more abstract, machine-independent description of the collection of computable partial functions.

We will eventually characterize it as the smallest collection containing some basic functions and closed under some fundamental operations for forming new functions from old, viz.

Composition, primitive recursion and minimization.

The characterization is due to Kleene (circa 1936), who made use of earlier, related work by Gödel and Herbrand.

We begin by looking at the basic functions, composition and primitive recursion; minimization will be considered in the section on partial recursive functions.

The basic functions :

Projection functions, $\mathrm{proj}^n_i \in \mathrm{Fun}(\mathbb{N}^n, \mathbb{N})$

$$\mathrm{proj}^n_i(x_1, \dots, x_n) \overset{\mathrm{def}}{=} x_i$$

Constant function with value zero, $\mathrm{zero}^n \in \mathrm{Fun}(\mathbb{N}^n, \mathbb{N})$

$$\mathrm{zero}^n(x_1, \dots, x_n) \overset{\mathrm{def}}{=} 0$$

Successor function, $\mathrm{suc} \in \mathrm{Fun}(\mathbb{N}, \mathbb{N})$

$$\mathrm{suc}(x) \overset{\mathrm{def}}{=} x+1$$

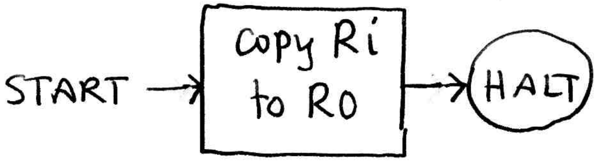
PROPOSITION :

The basic functions are computable.

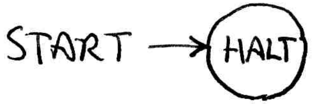
Proof

A register machine for computing

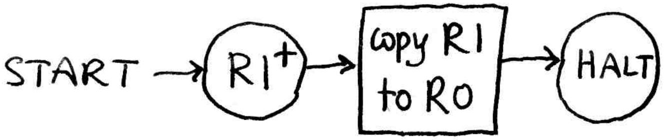
proj^n_i is specified by



zero^n " " "



suc " " "



□

Composition of partial functions

Given $f \in \mathrm{Pfn}(\mathbb{N}^n, \mathbb{N})$ & $g_1, \ldots, g_n \in \mathrm{Pfn}(\mathbb{N}^m, \mathbb{N})$, define $f \circ (g_1, \ldots, g_n) \in \mathrm{Pfn}(\mathbb{N}^m, \mathbb{N})$ by:

$f \circ (g_1, \ldots, g_n)(x_1, \ldots, x_m) = z \overset{\text{def}}{\Longleftrightarrow}$ there exists $(y_1, \ldots, y_n) \in \mathbb{N}^n$ so that

$$g_1(x_1, \ldots, x_m) = y_1 \ \& \cdots \& \ g_n(x_1, \ldots, x_m) = y_n$$

and $f(y_1, \ldots, y_n) = z$

Thus $f \circ (g_1, \ldots, g_n)$ is the unique m-ary partial function h satisfying $h(x_1, \ldots, x_m) \equiv f(g_1(x_1, \ldots, x_m), \ldots, g_n(x_1, \ldots, x_m))$

(where $\equiv$ denotes Kleene equivalence ...)

NOTE: in case $n = 1$, we write $f \circ g$ instead of $f \circ (g)$

Recall (from p.20) that for a partial function $h \in \mathrm{Pfn}(\mathbb{N}^m, \mathbb{N})$, we write "$h(x_1, \ldots, x_m) = z$" for "$((x_1, \ldots, x_m), z) \in h$", i.e. to mean that h is defined at $(x_1, \ldots, x_m)$ and takes value z there.

Thus "$h(x_1, \ldots, x_m)$" is an example of a partially defined expression, i.e. an expression which either denotes a specific value (a number, in this case) or is undefined.

"$f(g_1(x_1, \ldots, x_m), \ldots, g_n(x_1, \ldots, x_m))$" is a more complicated example of such partially defined expressions.

Given two partially defined expressions e and e', the statement

$$e \equiv e' \quad (\text{"}e \text{ and } e' \text{ are Kleene equivalent"})$$

is defined to mean

"either e and e' are both undefined, or they are both defined and the values they denote are equal"

Kleene equivalence allows one to express statements about undefinedness and equality in a convenient & succinct form.

PROPOSITION:

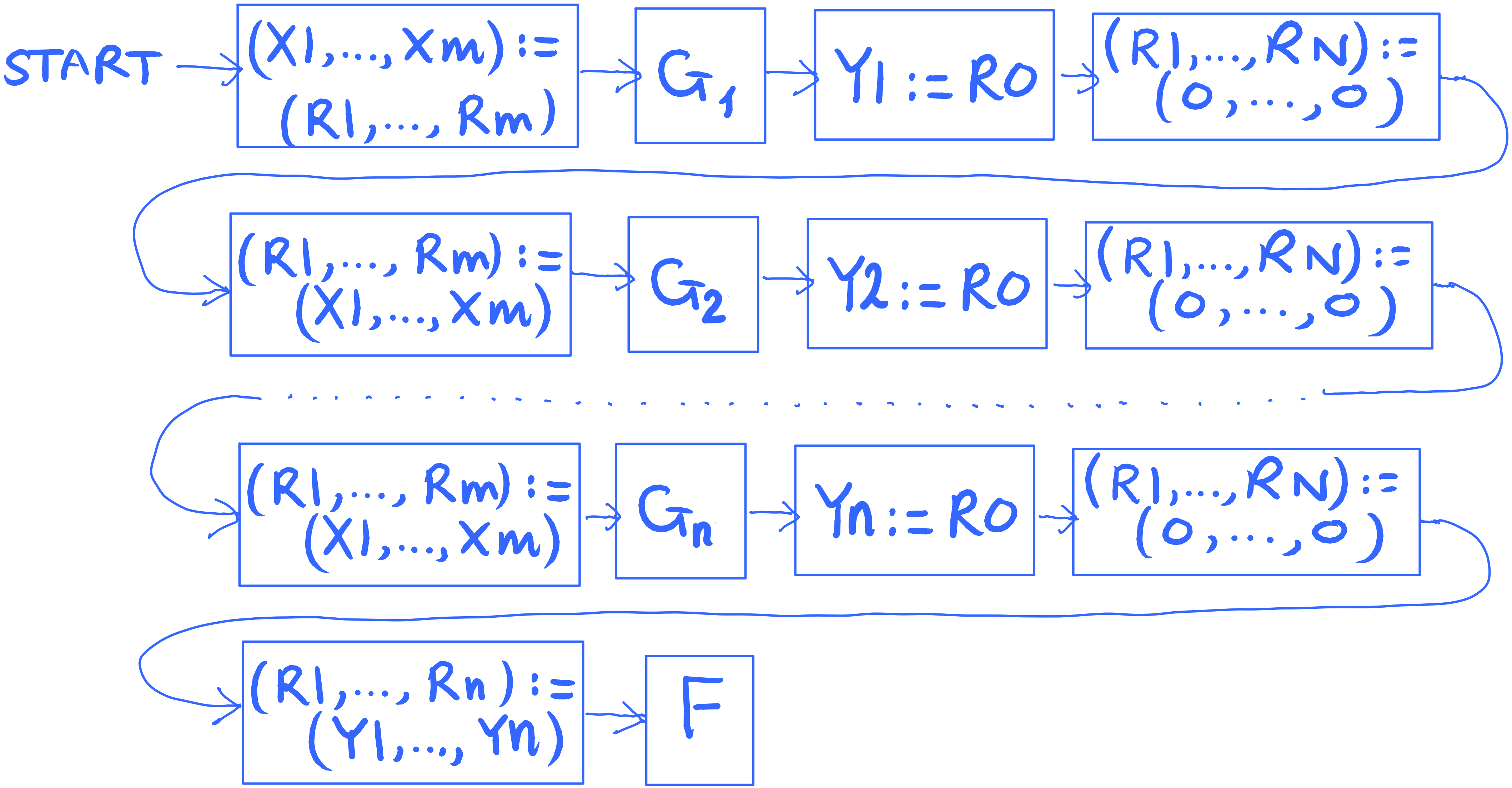
If f and $g_1,\dots,g_n$ are all computable, then so is $f\circ(g_1,\dots,g_n)$.

Proof.

Given register machine programs $\left\{\begin{matrix} F \\ G_i \end{matrix}\right.$ computing $\left\{\begin{matrix} f(y_1,\dots,y_n) \\ g_i(x_1,\dots,x_m) \end{matrix}\right.$ in RO starting with $\left\{\begin{matrix} R1,\dots,Rn \\ R1,\dots,Rm \end{matrix}\right.$ set to $\left\{\begin{matrix} y_1,\dots,y_n \\ x_1,\dots,x_m \end{matrix}\right.$, then the following graph specifies a program computing $f\circ(g_1,\dots,g_n)(x_1,\dots,x_m)$ in RO starting with R1,…,Rm set to $x_1,\dots,x_m$.

□

Program for $f\circ(g_1,\dots,g_n)$:



We assume programs F, G1,...,Gn only use registers R1,...,RN (where $N \geq \max\{n,m\}$) and that X1,...,Xm & Y1,...,Yn are not in that list.

To motivate the definition of primitive recursion, here are some examples of recursive definitions of partial functions $f \in \mathrm{Pfn}(\mathbb{N}, \mathbb{N})$

---

1. Sum of $0, 1, 2, \ldots, x$

$$\begin{cases} \mathrm{sum}(0) = 0 \\ \mathrm{sum}(x+1) = \mathrm{sum}(x) + x + 1 \end{cases}$$

---

2. nth Fibonacci number

$$\begin{cases} \mathrm{fib}(0) = 0 \\ \mathrm{fib}(1) = 1 \\ \mathrm{fib}(x+2) = \mathrm{fib}(x) + \mathrm{fib}(x+1) \end{cases}$$

---

3. A function that's undefined except when $x=0$

$$\begin{cases} f_3(0) = 0 \\ f_3(x+1) = f_3(x+2) + 1 \end{cases}$$

---

4. McCarthy's "91" function

$$f_4(x) = \text{if } x > 100 \text{ then } x-10$$
$$\text{else } f_4(f_4(x+11))$$

(f_4 maps x to 91 if $x \leq 100$
and to $x-10$ otherwise)

Primitive recursion

Given $f \in \mathrm{Pfn}(\mathbb{N}^n, \mathbb{N})$ and $g \in \mathrm{Pfn}(\mathbb{N}^{n+2}, \mathbb{N})$, define $\rho^n(f,g) \in \mathrm{Pfn}(\mathbb{N}^{n+1}, \mathbb{N})$ by

$\rho^n(f,g)(x_1,\ldots,x_n,x) = y \overset{\text{def}}{\iff}$ there exist $y_0, y_1, \ldots, y_x$ such that $f(x_1,\ldots,x_n) = y_0$

& $g(x_1,\ldots,x_n,i,y_i) = y_{i+1}$ for $i = 0,\ldots,x-1$

& $y_x = y$

It follows that $\rho^n(f,g)$ is the unique $(n+1)$-ary partial function h satisfying

$$\begin{cases} h(x_1,\ldots,x_n,0) \equiv f(x_1,\ldots,x_n) \\ h(x_1,\ldots,x_n,x+1) \equiv g(x_1,\ldots,x_n,x,h(x_1,\ldots,x_n,x)) \end{cases}$$

PROPOSITION :

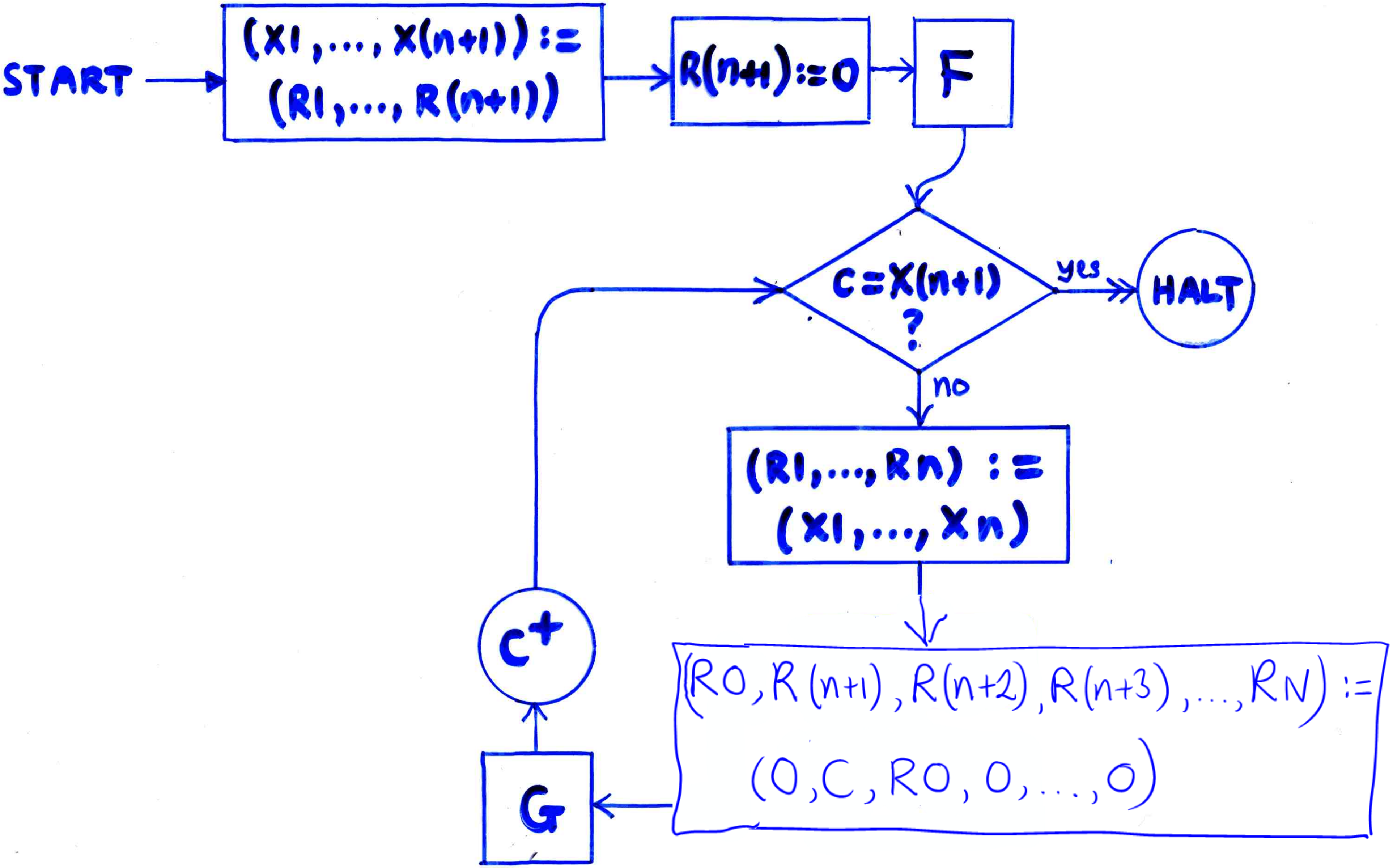
If f and g are computable, then so is $\rho^n(f,g)$.

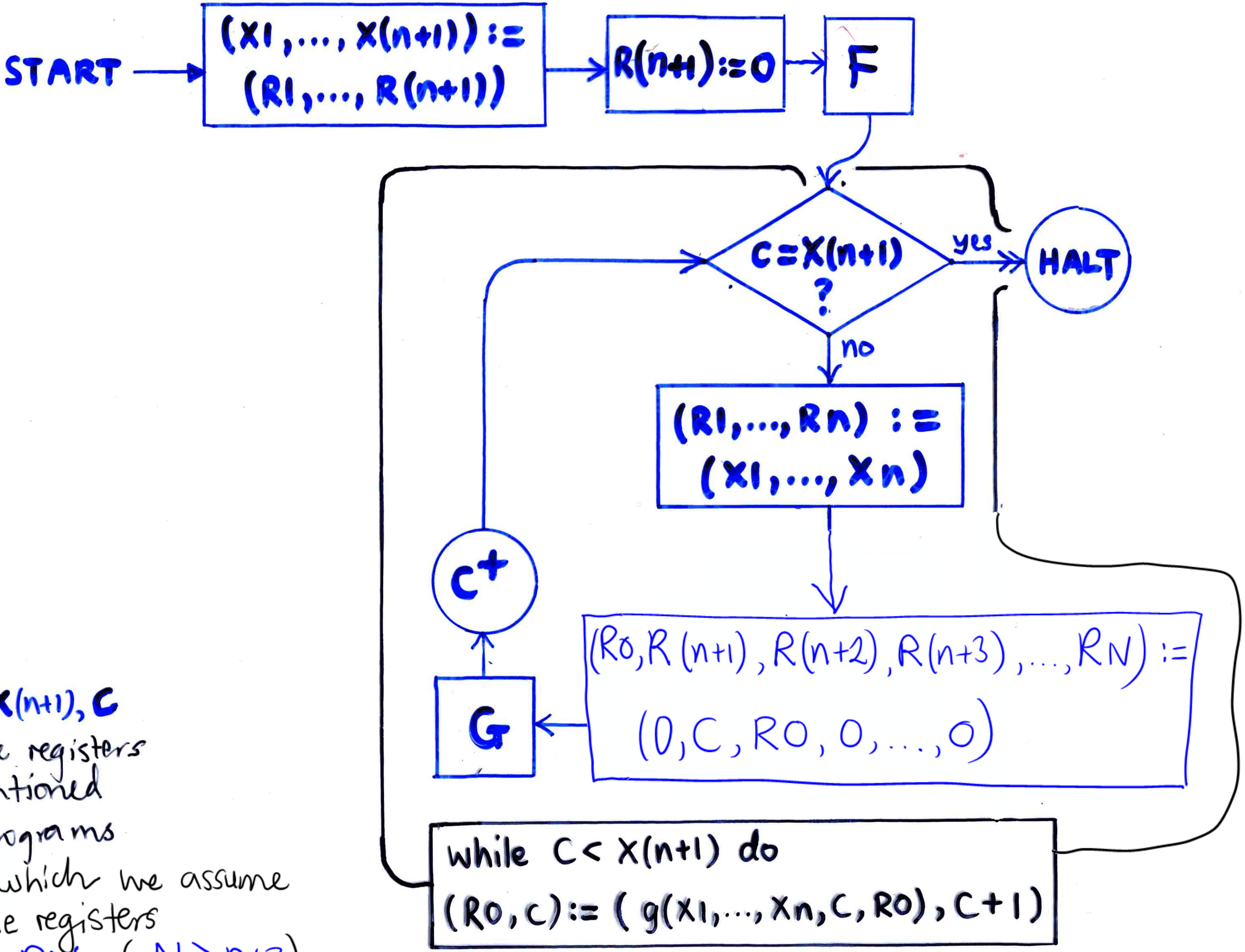
Proof

Given register machine programs

$$\begin{cases} F \\ G \end{cases} \text{computing} \begin{cases} f(x_1,\ldots,x_n) \\ g(x_1,\ldots,x_{n+2}) \end{cases} \text{in } R0 \text{ starting with} \begin{cases} R1,\ldots,Rn \\ R1,\ldots,R(n+2) \end{cases} \text{set to} \begin{cases} x_1,\ldots,x_n \\ x_1,\ldots,x_{n+2} \end{cases},$$

then the following diagram specifies a register machine program that computes $\rho^n(f,g)(x_1,\ldots,x_n)$ in R0 starting with R1,..., R(n+1) set to $x_1,\ldots,x_{n+1}$:


START
(X1,...,X(n+1)) := (R1,...,R(n+1))
R(n+1) := 0
F
C = X(n+1) ?
yes
HALT
no
(R1,...,Rn) := (X1,...,Xn)
(R0, R(n+1), R(n+2), R(n+3),...,RN) := (0, C, R0, 0,...,0)
G
C+



X1,…,X(n+1), C are some registers not mentioned in the programs **F** & **G**, which we assume only use registers R0,…,RN ($N \geq n+2$).

DEFINITION :

A function is <u>primitive recursive</u> if it can be built up from the basic functions by repeated use of the operations of composition and primitive recursion.

In other words, the set **PRIM** of primitive recursive functions is the <u>smallest</u> set of partial functions containing the basic functions and closed under the operations of composition and primitive recursion.

EXAMPLES of primitive recursive functions

Ex.1 Addition

Recall the inductive definition of $\text{add}(x,y) \overset{\text{def}}{=} x+y$ in terms of the successor function and zero:

$$\begin{cases} \text{add}(x,0) = x \\ \text{add}(x,y+1) = \text{add}(x,y)+1 \end{cases}$$

Thus $\text{add} = \rho^1(f,g)$ where $f(x) \overset{\text{def}}{=} x$
and $g(x,y,z) \overset{\text{def}}{=} z+1$.

Since $f = \text{proj}_1^1$ and $g = \text{suc} \circ \text{proj}_3^3$,

$$\text{add} = \rho^1(\text{proj}_1^1, \text{suc} \circ \text{proj}_3^3)$$

is primitive recursive.

Ex. 2 Multiplication $\text{mult}(x, y) \overset{\text{def}}{=} x.y$ can be inductively defined from addition by
$$\begin{cases} \text{mult}(x, 0) = 0 \\ \text{mult}(x, y+1) = \text{mult}(x, y) + x \end{cases}$$

Thus $\text{mult} = \rho^1(f, g)$ with $f(x) \overset{\text{def}}{=} 0$ and $g(x, y, z) \overset{\text{def}}{=} z + x$.

Hence $\text{mult} = \rho^1(\text{zero}^1, \text{add} \circ (\text{proj}^3_3, \text{proj}^3_1))$

$= \rho^1(\text{zero}^1, \rho^1(\text{proj}^1_1, \text{suc} \circ \text{proj}^3_3) \circ (\text{proj}^3_3, \text{proj}^3_1))$ by Ex. 1

is primitive recursive.

Ex. 3 Exponentiation $\exp(x, y) \overset{\text{def}}{=} x^y$ can be inductively defined from multiplication by
$$\begin{cases} \exp(x, 0) = 1 \\ \exp(x, y+1) = \exp(x, y).x \end{cases}$$

Thus $\exp = \rho^1(f, g)$ where $f(x) \overset{\text{def}}{=} 1$ and $g(x, y, z) \overset{\text{def}}{=} z.x$,

i.e. $f = \text{suc} \circ \text{zero}^1$ and $g = \text{mult} \circ (\text{proj}^3_3, \text{proj}^3_1)$.

Hence by Ex. 2,

$\exp = \rho^1(\text{suc} \circ \text{zero}^1, \rho^1(\text{zero}^1, \rho^1(\text{proj}^1_1, \text{suc} \circ \text{proj}^3_3) \circ (\text{proj}^3_3, \text{proj}^3_1)) \circ (\text{proj}^3_3, \text{proj}^3_1))$ [!!]

is primitive recursive.

[For the following examples, we leave as an exercise the working out of an explicit description of the function witnessing its primitive recursivity.]

Ex. 4 Predecessor function $\mathrm{pred}(x) \overset{\text{def}}{=} \begin{cases} 0 & \text{if } x = 0 \\ x-1 & \text{if } x > 0 \end{cases}$ is primitive recursive because it satisfies $\begin{cases} \mathrm{pred}(0) = 0 \\ \mathrm{pred}(x+1) = x \end{cases}$.

Ex. 5 Truncated subtraction $\mathrm{minus}(x,y) \overset{\text{def}}{=} x \dot{-} y = \begin{cases} 0 & \text{if } x < y \\ x-y & \text{if } x \geq y \end{cases}$

satisfies $\begin{cases} \mathrm{minus}(x,0) = x \\ \mathrm{minus}(x,y+1) = \mathrm{pred}(\mathrm{minus}(x,y)) \end{cases}$ and hence is prim. rec. by Ex. 4.

Ex. 6 Conditional function $\mathrm{ifzero}(x,y,z) \overset{\text{def}}{=} \begin{cases} y & \text{if } x = 0 \\ z & \text{if } x > 0 \end{cases}$

Note that $\mathrm{ifzero} = C \circ (\mathrm{proj}^3_2, \mathrm{proj}^3_3, \mathrm{proj}^3_1)$, where $C \in \mathrm{Fun}(\mathbb{N}^3, \mathbb{N})$ satisfies $\begin{cases} C(x_1, x_2, 0) = x_1 \\ C(x_1, x_2, x+1) = x_2 \end{cases}$.

Thus C, and hence also ifzero, are primitive recursive.

Ex.7 Bounded Summation

If $f \in \mathrm{Fun}(\mathbb{N}^{n+1}, \mathbb{N})$ is prim. rec., then so is

$$\boxed{\begin{aligned} g(x_1,\dots,x_n,x) &\triangleq \sum_{y<x} f(x_1,\dots,x_n,y) \\ &= \begin{cases} f(x_1,\dots,x_n,0)+\cdots+f(x_1,\dots,x_n,x-1) & \text{if } x>0 \\ 0 & \text{if } x=0 \end{cases} \end{aligned}}$$

(For note that g satisfies

$$\begin{cases} g(x_1,\dots,x_n,0) = 0 \\ g(x_1,\dots,x_n,x+1) = g(x_1,\dots,x_n,x)+f(x_1,\dots,x_n,x) \end{cases} .)$$

with f prim. rec (by assumption) & add prim. rec. by Ex. 1

PROPOSITION:

Every primitive recursive function is both computable and total

(Recall that $f \in \mathrm{Pfn}(\mathbb{N}^n, \mathbb{N})$ is total if & only if $f(x_1, \dots, x_n)\downarrow$ for all $(x_1, \dots, x_n) \in \mathbb{N}^n$.)

NOTE that by definition of PRIM (see p.94), if P is some property of partial functions, to prove that every member of PRIM has property P, it suffices to show that

(a) the basic functions (proj_i^n, zero^n, suc) satisfy P ; and

(b) if $f, g_1, \ldots, g_n$ satisfy P, then so does $f \circ (g_1, \ldots, g_n)$ [assuming of course that the functions have arities for which the composition makes sense] ; and

(c) if f & g satisfy P, then so does $\rho^n(f, g)$ [where n = arity of f & g has arity n+2].

For if (a), (b) & (c) hold, then $\{f \in \mathrm{PRIM} \mid f \text{ satisfies } P\}$ contains the basic functions and is closed under composition and primitive recursion, and hence contains all primitive recursive functions : so they all satisfy P.

Proof of the Proposition on p.99

We have already verified that (a), (b) & (c) hold when P is the property "f is computable" : hence every $f \in PRIM$ is computable.

Now taking P to be the property "f is a total function", clearly (a) & (b) hold ; and to see that (c) holds, note that if f & g are total then for all $(x_1, \ldots, x_n) \in \mathbb{N}^n$ the definition of $\rho^n(f,g)$ gives

$$\rho^n(f,g)(x_1, \ldots, x_n, 0) \downarrow$$

and $$\rho^n(f,g)(x_1, \ldots, x_n, x) \downarrow \Rightarrow \rho^n(f,g)(x_1, \ldots, x_n, x+1) \downarrow$$

so that $\forall x . \rho^n(f,g)(x_1, \ldots, x_n, x)$ by Mathematical Induction on x. Hence every $f \in PRIM$ is total. $\square$

Partial recursive functions

We saw above that

primitive recursive ⇒ computable & total

Since not every computable partial function is total, it is certainly not the case that every computable partial function is primitive recursive.

One intuitively algorithmic method of calculation that can give rise to non-total partial functions is that of searching for the smallest value of a function's argument that produces a given result (zero, say) – since no such value may exist. The corresponding operation on functions is called minimization…

Minimization

Given $f \in \mathrm{Pfn}(\mathbb{N}^{n+1}, \mathbb{N})$

define $\mu(f) \in \mathrm{Pfn}(\mathbb{N}^n, \mathbb{N})$ by

$$\mu(f)(x_1,\ldots,x_n) = x \overset{\text{def}}{\iff} \text{there exist } y_0, y_1, \ldots, y_x$$

such that $f(x_1,\ldots,x_n,i) = y_i$ for $i = 0,1,\ldots,x$

& $y_i > 0$ for $i = 0,1,\ldots,x-1$

& $y_x = 0$

Thus

$$\mu(f)(x_1,\ldots,x_n) \equiv \begin{cases} \text{the least } x \text{ such that} \\ f(x_1,\ldots,x_n,x) = 0 \text{ and} \\ f(x_1,\ldots,x_n,i) > 0 \text{ for } i < x \end{cases}$$

(in particular, $f(x_1,\ldots,x_n,i)\downarrow$ for $i<x$)

and in particular

$\mu(f)(x_1,\ldots,x_n)\uparrow$ if no x exists satisfying these conditions.

PROPOSITION:

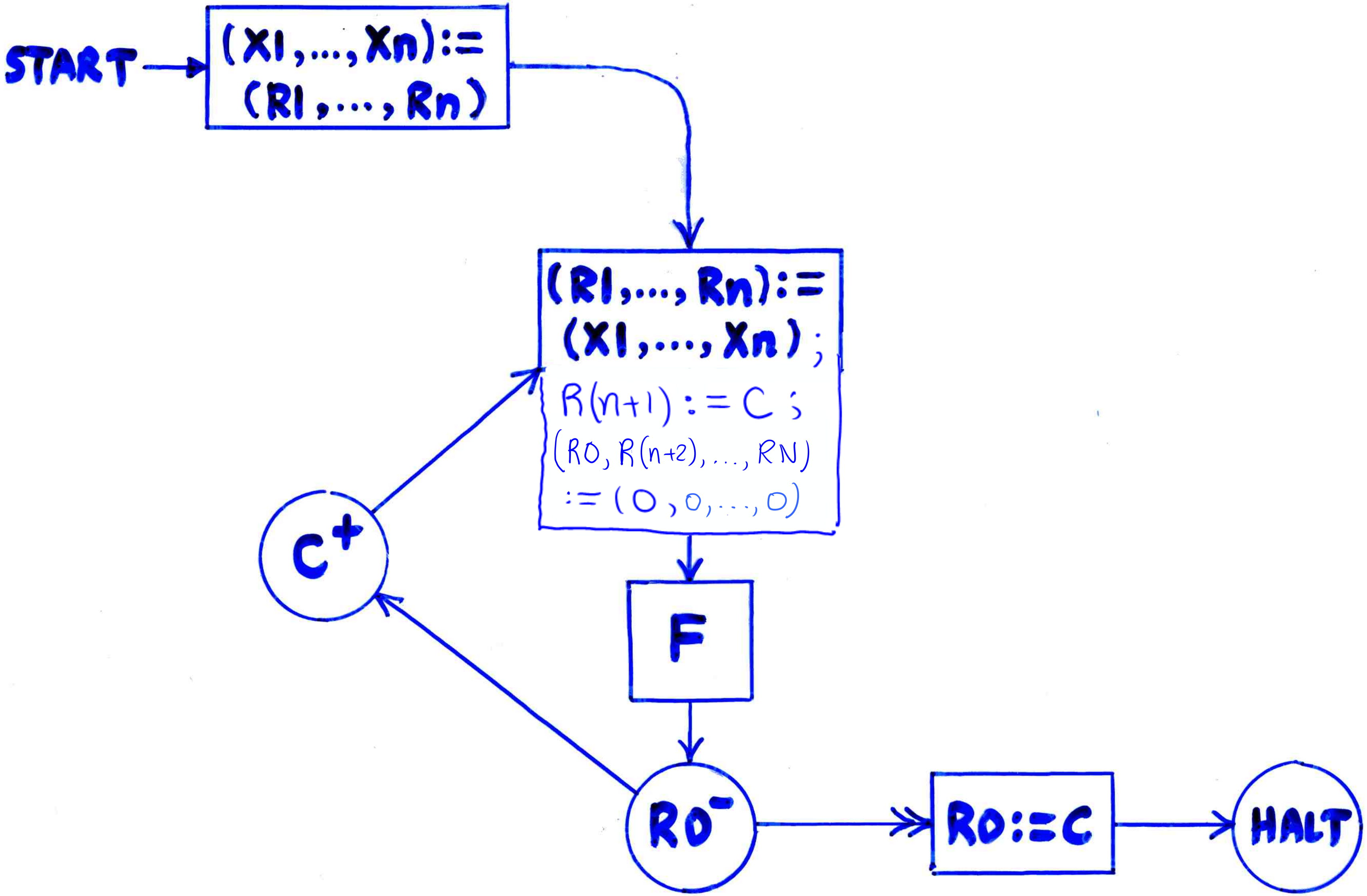
If f is computable, then so is $\mu(f)$.

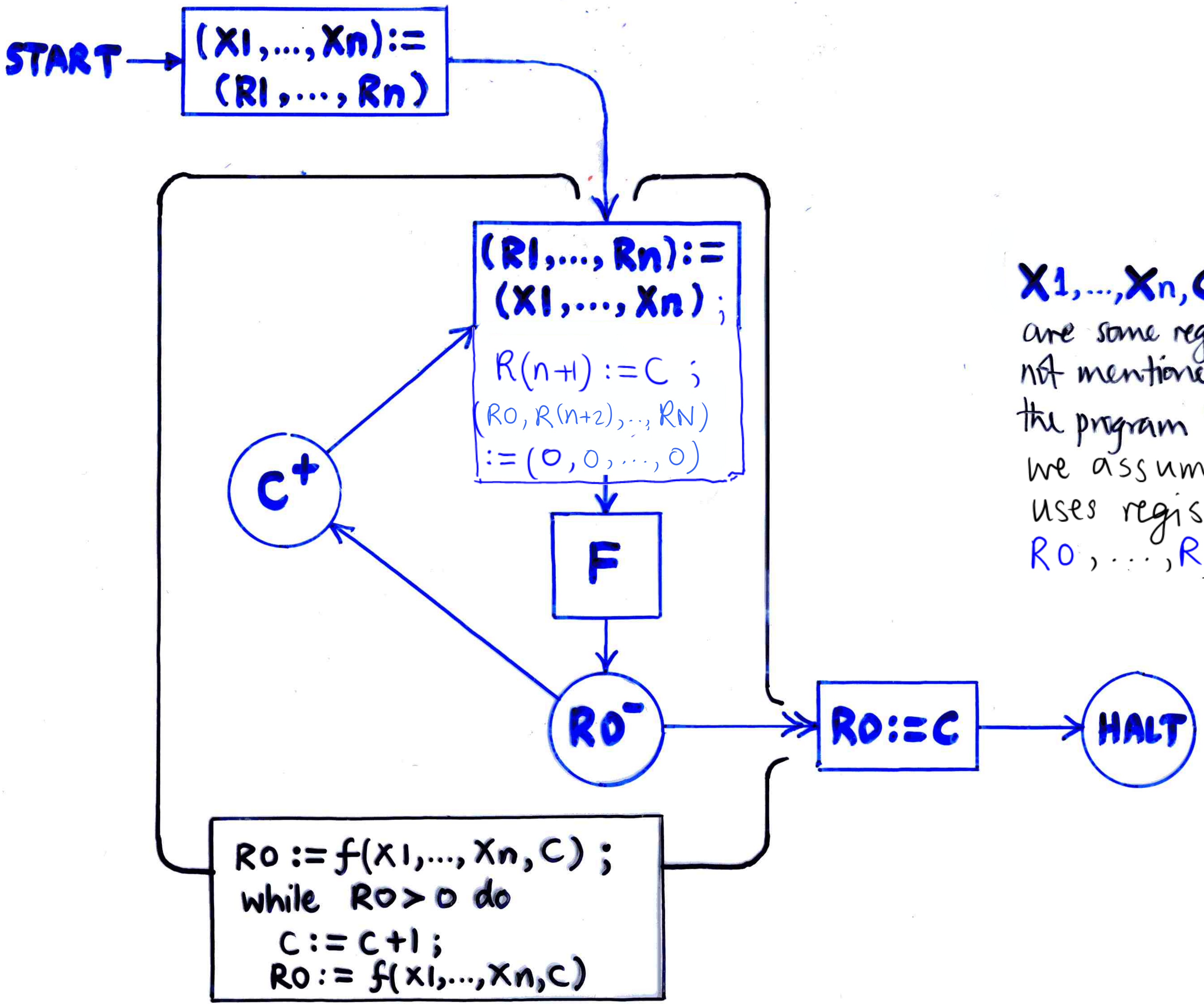
Proof

Given a register machine program

F computing $f(x_1, \ldots, x_{n+1})$ in R0 starting with R1, …, R(n+1) set to $x_1, \ldots, x_{n+1}$

then the following diagram specifies a register machine program for computing $\mu(f)(x_1, \ldots, x_n)$ in R0 starting with R1, …, Rn set to $x_1, \ldots, x_n$:


START
(X1,...,Xn):=
(R1,...,Rn)
(R1,...,Rn):=
(X1,...,Xn);
R(n+1) := C ;
(R0, R(n+2), ..., RN)
:= (0, 0, ..., 0)
C+
F
R0-
R0:=C
HALT



X1,...,Xn, C are some registers not mentioned in the program **F**, which we assume only uses registers R0,...,RN (N > n).

DEFINITION :

A partial recursive function is a partial function that can be built up from the basic functions by repeated use of the operations of composition, primitive recursion and minimization.

In other words, the set $\boxed{PR}$ of partial recursive functions is the smallest set of partial functions containing the basic functions and closed under the operations of composition, primitive recursion and minimization.

The members of PR that are total are called (total) recursive functions.

EXAMPLES of minimization

Ex. 1 The everywhere undefined function is partial recursive. For if $f(x,y) \overset{\text{def}}{=} 1$, then $\mu(f)(x)\uparrow$, for all x; and $f = \text{suc} \circ \text{zero}^2$; so $\mu(f) = \mu(\text{suc} \circ \text{zero}^2)$ is partial recursive.

Ex. 2

$$\left.\begin{array}{l} d(x,y) \overset{\text{def}}{=} \text{integer part of } x/y \text{ (undefined if } y=0) \\ m(x,y) \overset{\text{def}}{=} \text{remainder when } x \text{ is divided by } y \end{array}\right\} \text{are partial recursive.}$$

For note that

$$d(x,y) \equiv \text{least } z \text{ such that } x < y.(z+1) \qquad (\text{thus } d(x,0)\uparrow).$$

Now $ge(x,y) \overset{\text{def}}{=} \begin{cases} 0 & \text{if } x < y \\ 1 & \text{if } x \geq y \end{cases}$ is primitive recursive, since

$ge(x,y) = \text{ifzero}(y \dot{-} x, 1, 0)$ (and we saw above that ifzero & $\dot{-}$ are in PRIM).

So $f(x,y,z) \overset{\text{def}}{=} ge(x, y.(z+1))$ is also in PRIM (since multiplication is)

Then $d = \mu(f)$ is partial recursive.

Finally, note that $m(x,y) \equiv x \dot{-} y.d(x,y)$, so m is also in PR.

N.B.

The function d in Ex. 2 is not total recursive because it is undefined when its second argument is 0.

Thus

$$\mathrm{div}(x,y) \overset{\mathrm{def}}{=} \begin{cases} \text{integer part of } x/y & \text{if } y>0 \\ 0 & \text{if } y=0 \end{cases}$$

is (total) recursive : $\mathrm{div} = \mathrm{ifzero} \circ (\mathrm{proj}^2_2, \mathrm{zero}^2, d)$.

In fact, div is primitive recursive (exercise: prove this).

Every $f \in$ PRIM satisfies

$f \in$ PR & f is total

BUT converse is false :

there are total recursive functions which are not primitive recursive

Here is a sketch of the proof of this, making use of our next major result — the Theorem on page 114 ...

Proof (sketch)

First,

(eg $\rho^1(\mathrm{proj}^1_1, \mathrm{suc} \circ \mathrm{proj}^3_3)$ is a formal description for $\mathrm{add}(x,y) \overset{\mathrm{def}}{=} x+y$)

code $\boxed{\text{formal descriptions}}$ of primitive recursive functions as numbers so that

$$e(x,y) \overset{\mathrm{def}}{=} \begin{cases} f_x(y) & \text{if } x \text{ is code of a formal description of a unary prim. rec. function, } f_x \text{ say} \\ 0 & \text{if } x \text{ is not the code of a formal description of a unary prim. rec. function} \end{cases}$$

is a computable function.

Next,

[This is a similar diagonalization trick to that in the proof that not all functions are computable

consider $e'(x) \overset{\mathrm{def}}{=} e(x,x)+1$

CLAIM : $e' \in PR$, but $e' \notin PRIM$.

Next we make use of the Theorem to be proved below (p114), namely that PR coincides with the collection of computable functions. Since e is computable, this Theorem implies that it is in PR — and hence so is e' since

$$e' = \mathrm{suc} \circ (e \circ (\mathrm{proj}^1_1, \mathrm{proj}^1_1)).$$

To see that $e' \notin \mathrm{PRIM}$, suppose the contrary and derive a contradiction: if $e' \in \mathrm{PRIM}$, then it would have a formal description, and hence $e' = f_x$ for some code x. Then

$$\begin{aligned} f_x(x) &= e'(x) && \text{since } e' = f_x \\ &= e(x,x)+1 && \text{by definition of } e' \\ &= f_x(x)+1 && \text{by definition of } e \end{aligned}$$

which is impossible! $\square$

Here is a more explicit example of a non-primitive-recursive member of PR:

Ackermann's function

FACT : there is a total function $\mathrm{ack} \in \mathrm{Fun}(\mathbb{N} \times \mathbb{N}, \mathbb{N})$ satisfying

$$\begin{cases} \mathrm{ack}(0, y) = y+1 \\ \mathrm{ack}(x+1, 0) = \mathrm{ack}(x, 1) \\ \mathrm{ack}(x+1, y+1) = \mathrm{ack}(x, \mathrm{ack}(x+1, y)) \end{cases}$$

ack is recursive, but not primitive recursive.

It is beyond the scope of this course to prove that $ack \notin PRIM$.
(Roughly speaking, $ack \notin PRIM$ because as x & y increase, $ack(x,y)$ grows faster than any primitive recursive function possibly can grow.)
One way to see that $ack \in PR$ is to design a register machine to compute ack (exercise), and then appeal to the Theorem we are about to prove that states that PR coincides with the set of computable functions.
The proof that the register machine for ack always halts (i.e. that ack is total) is non-trivial. (It can be done by "well-founded induction" on pairs $(x,y) \in \mathbb{N}^2$ ordered lexicographically : $(x_1,y_1)<(x_2,y_2)$
$\Leftrightarrow (x_1 < x_2$ or $(x_1 = x_2$ & $y_1 < y_2))$.)

THEOREM:

A partial function is (register machine) computable if & only if it is partial recursive.

We have already proved that the collection of computable partial functions contains the basic functions and is closed under the operations of composition, primitive recursion and minimization, and hence contains all partial recursive functions.

So it remains to see that

$$f \text{ computable} \Rightarrow f \text{ partial recursive}$$

Proof of (f computable $\Rightarrow f \in PR$)

If $f \in Pfn(\mathbb{N}^n, \mathbb{N})$ is computable, there is a register machine M which when started with R1, ..., Rn set to $x_1, \dots, x_n$ (and all other registers set to 0), halts if & only if $f(x_1, \dots, x_n)\downarrow$, and in that case R0 contains this value.

Suppose the registers of M are R0, R1, R2, ..., Rn, R(n+1), ..., Rm (for some $m \geq n$).
Suppose M's program has instructions labelled L0, L2, ..., LI (some $I \geq 0$), and without loss of generality assume that the only HALT instruction is the last one (LI) and that there are no erroneous halts (i.e. the only labels referred to in increment/decrement instructions lie in the range L0, ..., LI).
The <u>state</u> of M at any stage in its computation can be specified by the code $[\ell, r_0, r_1, \dots, r_m]$ of a list of length $m+2$, where

ℓ = current instruction (so $0 \leq \ell \leq I$)
r_j = current contents of Rj ($j = 0, 1, \dots, m$).

The proof that f is partial recursive depends upon the following lemmas :

LEMMA 1 :

There are primitive recursive functions $lab, val_0, val_1, \ldots, val_m \in Fun(\mathbb{N}, \mathbb{N})$ satisfying

$$\begin{cases} lab([\ell, r_0, \ldots, r_m]) = \ell \\ val_j([\ell, r_0, \ldots, r_m]) = r_j \end{cases} \qquad \left(\begin{array}{l}\text{for all } j = 0, \ldots, m \text{ and} \\ \text{all } (\ell, r_0, \ldots, r_m) \in \mathbb{N}^{m+2}\end{array}\right)$$

Thus lab gives the label of a state of M, whilst val_j gives the value held in R_j.

LEMMA 2 :

There is a primitive recursive function $next \in Fun(\mathbb{N}, \mathbb{N})$ which gives the next state of M in terms of the current one.

The proof of these lemmas uses the following property of our coding of lists of numbers as numbers ...

PROPOSITION:

The functions

$\mathrm{mklist}^n \in \mathrm{Fun}(\mathbb{N}^n, \mathbb{N})$

$\mathrm{hd}, \mathrm{tl} \in \mathrm{Fun}(\mathbb{N}, \mathbb{N})$

defined by

$$\mathrm{mklist}^n(x_1, \dots, x_n) \overset{\text{def}}{=} [x_1, \dots, x_n]$$

$$\mathrm{hd}(x) \overset{\text{def}}{=} \begin{cases} x_1 & \text{if } x = [x_1, \dots, x_n] \text{ for some } n > 0 \;\&\; x_1, \dots, x_n \\ 0 & \text{if } x = [\mathrm{nil}] = 0 \end{cases}$$

$$\mathrm{tl}(x) \overset{\text{def}}{=} \begin{cases} [x_2, \dots, x_n] & \text{if } x = [x_1, \dots, x_n] \text{ for some } n > 0 \;\&\; x_1, \dots, x_n \\ 0 & \text{if } x = [\mathrm{nil}] = 0 \end{cases}$$

are all primitive recursive.

Proof of (f computable $\Rightarrow$ f $\in$ PR), cont.

First, note that

$$\text{state}(x_1,\dots,x_n,t) \overset{\text{def}}{=} \left\{ \begin{array}{l} \text{state of } M \text{ at } t^{\text{th}} \text{ step,} \\ \text{starting with } R1=x_1,\dots, Rn=x_n \\ \text{\& all other registers} = 0 \end{array} \right.$$

is primitive recursive, because

$$\left\{ \begin{array}{l} \text{state}(x_1,\dots,x_n,0) = [0,0,x_1,\dots,x_n,0,\dots,0] \\ \\ \text{state}(x_1,\dots,x_n,t+1) = \text{next}(\text{state}(x_1,\dots,x_n,t)) \end{array} \right.$$

so that

$$\text{state} = \rho^n(\text{mklist}^{m+2} \circ (\text{zero}^n, \text{zero}^n, \text{proj}^n_1, \dots, \text{proj}^n_n, \text{zero}^n, \dots, \text{zero}^n), \; \text{next} \circ \text{proj}^{n+2}_{n+2})$$

with mklist^{m+2}, next $\in$ PRIM.

Now:

$$f(x_1,\ldots,x_n) \equiv val_0(state(x_1,\ldots,x_n,halt(x_1,\ldots,x_n)))$$

where

$$halt(x_1,\ldots,x_n) \overset{def}{=} \begin{cases} \text{number of steps taken to halt} \\ \text{(\& undefined if never halt)} \end{cases}$$

$$\equiv \text{least } t \text{ such that } lab(state(x_1,\ldots,x_n,t)) = I$$

$$\equiv \mu(h)(x_1,\ldots,x_n)$$

where $h(x_1,\ldots,x_n,t) \overset{def}{=} I \mathbin{\dot{-}} lab(state(x_1,\ldots,x_n,t))$.

Since lab, state & $\dot{-}$ are in PRIM, so is h and hence $halt = \mu(h)$ is in PR.

Thus $f = val_0 \circ (state \circ (proj^n_1,\ldots,proj^n_n, \mu(h)))$ is also in PR.

□

To complete the proof of the Theorem, we have to prove the Proposition and Lemmas 1 & 2.

Proof of the Proposition

Since $\mathrm{mklist}^n(x_1, \ldots, x_n) = \langle x_1, \langle x_2, \ldots \langle x_n, 0 \rangle \cdots \rangle \rangle$, to see that $\mathrm{mklist}^n \in \mathrm{PRIM}$, it suffices to show that $\langle x, y \rangle = 2^x(2y+1)$ is primitive recursive — which follows from the fact that multiplication and exponentiation are in PRIM.

The proof that hd and tl are in PRIM requires more effort. We get to their primitive recursivity via that of a number of intermediate functions:

(i) $\mathrm{mod}_2(x) \overset{\text{def}}{=} \begin{cases} 0 & \text{if } x \text{ even} \\ 1 & \text{if } x \text{ odd} \end{cases}$ is primitive recursive, because

it satisfies $\begin{cases} \mathrm{mod}_2(0) = 0 \\ \mathrm{mod}_2(x+1) = \mathrm{ifzero}(\mathrm{mod}_2(x), 1, 0) \end{cases}$.

(ii) $\mathrm{half}(x) \overset{\text{def}}{=}$ integer part of $x/2$ is primitive recursive by (i), since

it satisfies $\begin{cases} \mathrm{half}(0) = 0 \\ \mathrm{half}(x+1) = \mathrm{ifzero}(\mathrm{mod}_2(x), \mathrm{half}(x), \mathrm{half}(x)+1) \end{cases}$.

(iii)

$$f(x,y) \overset{\text{def}}{=} \begin{cases} x/2^y & \text{if } x>0 \text{ \& } 2^y \text{ divides } x \\ 0 & \text{otherwise} \end{cases}$$

is primitive recursive by (i) & (ii), since it satisfies

$$\begin{cases} f(x,0) = x \\ f(x,y+1) = \text{ifzero}(\text{mod}_2(f(x,y)), \text{half}(f(x,y)), 0) \end{cases}.$$

Combining (ii), (iii), (iv), and the fact (cf. page 98) that bounded summations preserve the property of primitive recursiveness, we have that hd and tl are in PRIM because ...

$$\mathrm{hd}(x) = \begin{cases} \text{largest } y \text{ such that } 2^y \text{ divides } x, & \text{if } x > 0 \\ 0, & \text{if } x = 0 \end{cases}$$

$$= \sum_{y<x} \mathrm{ifzero}(f(x, y+1), 0, 1)$$

and

$$\mathrm{tl}(x) = \mathrm{half}(f(x, \mathrm{hd}(x)))$$

☐ Proposition

Proof of Lemma 1

This follows immediately from the Proposition, because

$$\mathrm{lab} = \mathrm{hd}$$

and $$\mathrm{val}_j = \mathrm{hd} \circ (\underbrace{\mathrm{tl} \circ \cdots \circ \mathrm{tl}}_{j+1}).$$

$\square$ Lemma 1

Proof of Lemma 2

By examining M's program, we can define four $(I+1)$-tuples of numbers

$(a_0, \ldots, a_I)$, $(b_0, \ldots, b_I)$, $(c_0, \ldots, c_I)$, and $(d_0, \ldots, d_I)$

as follows:

- for each $i = 0, \ldots, I-1$
  if i^{th} instruction is an increment, say $Li : Rj^+ \to Lk$,
  then define $a_i = j$, $b_i = 0$, $c_i = k$, and $d_i = k$,
  else the instruction is a decrement, say $Li : Rj^- \to Lk, L\ell$,
  and define $a_i = j$, $b_i = 1$, $c_i = k$, and $d_i = \ell$
- define $a_I = 0$, $b_I = 0$, $c_i = I$, and $d_i = I$.

Summary:

If i^{th} instruction is,	then $(a_i, b_i, c_i, d_i) \overset{def}{=}$
$L_i : R_j^+ \to L_k$	$(j, 0, k, k)$
$L_i : R_j^- \to L_k, L_\ell$	$(j, 1, k, \ell)$
$L_I : \text{HALT}$	$(0, 0, I, I)$

$$next_\ell(x) \overset{def}{=} \sum_{i=0}^{I} ifzero(val_{a_i}(x), d_i, c_i) . eq(i, lab(x))$$

$$next_j(x) \overset{def}{=} \sum_{i=0}^{I-1} f_{ij}(x) . eq(i, lab(x)) + val_j(x) . eq(I, lab(x))$$

Where

$$f_{ij}(x) \overset{def}{=} \Big(b_i (val_j(x) \dot{-} 1) + (1 - b_i)(val_j(x) + 1) \Big) . eq(a_i, j) + val_j(x) . (1 - eq(a_i, j))$$

Note that the equality test function $eq(x,y) \overset{def}{=} \begin{cases} 1 & \text{if } x=y \\ 0 & \text{if } x \neq y \end{cases}$

is primitive recursive, since $eq(x,y) = ifzero(x \dot{-} y, ifzero(y \dot{-} x, 1, 0), 0)$. Using it, we can define primitive recursive functions nextl and $next_j$ as follows:

$$nextl(x) \overset{def}{=} \sum_{i=0}^{I} ifzero(val_{a_i}(x), d_i, c_i).eq(i, lab(x))$$

$$next_j(x) \overset{def}{=} \sum_{i=0}^{I-1} f_{ij}(x).eq(i, lab(x)) + val_j(x).eq(I, lab(x))$$

where

$$f_{ij}(x) \overset{def}{=} \big(b_i . pred(val_j(x)) + (1-b_i).suc(val_j(x))\big).eq(a_i, j) + val_j(x).(1-eq(a_i,j))$$

(and recall that $suc(x) = x+1$, $pred(x) = x \dot{-} 1$).

By choice of the constants a_i, b_i, c_i, d_i $(i=0,\ldots,I)$, it follows that given a state $[\ell, r_0, \ldots, r_m]$ of M,

$nextl([\ell, r_0, \ldots, r_m])$ = number of the instruction in the next state

$next_j([\ell, r_0, \ldots, r_m])$ = contents of R_j in the next state

(provided $0 \leq j \leq m$).

Therefore the next-state function is given by

$$\mathrm{next}(x) = [\,\mathrm{nextl}(x), \mathrm{next}_0(x), \ldots, \mathrm{next}_m(x)\,]$$

i.e.

$$\mathrm{next} = \mathrm{mklist}^{m+2} \circ (\mathrm{nextl}, \mathrm{next}_0, \ldots, \mathrm{next}_m)$$

and hence it is primitive recursive since nextl, next_i, and (by the Proposition) mklist^{m+2} are all in PRIM.

☐ Lemma 2

Recursive and recursively enumerable sets

So far we have concentrated on the aspect of algorithms to do with computing functions from inputs to outputs.

Another important use of algorithms is to generate, or enumerate, the elements of some set of data.

One says that a set S is effectively enumerable if there is some algorithm A which lists the elements of S:

$$S = \{ A(0), A(1), A(2), \dots \}$$

(It may well be that an element of S occurs many times in the list, but no matter.)

EXAMPLE:

The set PR of partial recursive functions is effectively enumerated by the algorithm

A which, given input x,

decodes x as a pair $x = \langle n, e\rangle$, then

decodes e as a register machine program $Prog_e$,

and returns the n-ary computable (hence partial recursive) function $\varphi_e^{(n)}$, where

$$\varphi_e^{(n)}(x_1, \ldots, x_n) = y \overset{\text{def}}{\iff} \begin{array}{l}\text{computation of } Prog_e \text{ started} \\ \text{with } R1, \ldots, Rn \text{ set to } x_1, \ldots, x_n \\ \text{halts with } R0 = y\end{array}$$

(because <u>every</u> element of PR is of the form $\varphi_e^{(n)}$ for some n & e)

Clearly, S has to be a countable set if it is effectively enumerable.

[Recall :

S is <u>countably infinite</u> if there is some bijection (= one-one and onto function) between $\mathbb{N}$ and S.

S is <u>countable</u> if it is either finite or countably infinite.

S is <u>uncountable</u> if it is not countable.

E.g. Fun($\mathbb{N}$, $\mathbb{N}$) is uncountable, by Cantor's Diagonal Argument.]

The notion of "effective enumerability" is an informal one, because it refers to the informal notion of "algorithm". We can formalize it using the notion of computable (= partial recursive) function provided we identify the set S to be enumerated with a subset of $\mathbb{N}$.... (Since S is necessarily countable, we can always do this some way).

DEFINITION:

A subset $S \subseteq \mathbb{N}$ of numbers is recursively enumerable (or r.e., for short) if & only if either it is empty ($S = \emptyset$) or there is a (total) recursive function $f \in \mathrm{Fun}(\mathbb{N}, \mathbb{N})$ so that

$$S = \{ f(n) \mid n \in \mathbb{N} \}$$

Recall: $S \subseteq \mathbb{N}$ is decidable if & only if the characteristic function of S

$$\chi_S(x) \overset{\text{def}}{=} \begin{cases} 1 & \text{if } x \in S \\ 0 & \text{if } x \notin S \end{cases}$$

is computable. (cf. p55)

Such sets are also called recursive (since χ_S is computable if & only if it is recursive, being a total function).

PROPOSITION:

Every recursive set is recursively enumerable.

Proof

Suppose S is recursive. If $S = \emptyset$, then S is r.e. by definition; otherwise we can find some $x_0 \in S$. Then since χ_S is recursive, so is

$$f(x) \overset{\text{def}}{=} \mathrm{ifzero}(\chi_S(x), x_0, x)$$

and $S = \{ f(x) \mid x \in \mathbb{N} \}$, so S is r.e. $\square$

In the section on the Halting Problem we saw that the set

$$\{ e \in \mathbb{N} \mid \varphi_e \text{ is a total function} \}$$

is undecidable. In fact it is not even recursively enumerable ...

EXAMPLE of a non-r.e. set

$$\mathrm{TOT} \overset{\text{def}}{=} \{ e \in \mathbb{N} \mid \varphi_e \text{ is a total function} \}$$

is not recursively enumerable.

Proof

If TOT were r.e., then (since $\mathrm{TOT} \neq \emptyset$) $\mathrm{TOT} = \{ f(x) \mid x \in \mathbb{N} \}$ for some recursive function $f \in \mathrm{Fun}(\mathbb{N}, \mathbb{N})$.

Let $u \in \mathrm{Pfn}(\mathbb{N}^2, \mathbb{N})$ be the partial function $u(e,x) \overset{\text{def}}{=} \varphi_e(x)$

CLAIM

(1) u is partial recursive; hence so is $g(x) \overset{\text{def}}{=} u(f(x), x) + 1$

(2) g is total; hence $g = \varphi_e$ for some $e \in \mathrm{TOT}$, but

(3) $e \neq f(x)$ for any $x \in \mathbb{N}$ – contradiction!

Proof of the CLAIMS:

(1) follows from the work we did in the section on a universal register machine $\mathcal{U}$, since $u(e,x)$ is the result (if any) of running $\mathcal{U}$ starting with $P = e$ and $A = [x]$.
Thus u is computable, and hence is partial recursive.

(2) Since by assumption on f, for all $x \in \mathbb{N}$ $f(x) \in TOT$ so $\varphi_{f(x)}(x)\downarrow$, so $g(x)\downarrow$ (by definition of g). Thus g is total recursive, and hence $g = \varphi_e$ for some $e \in TOT$.

(3) If $e = f(x)$, then

$$\begin{aligned} g(x) &= \varphi_e(x) && \text{since } g = \varphi_e \\ &= u(e,x) && \text{by definition of } u \\ &\neq u(e,x)+1 && \text{since } u(e,x) = g(x)\downarrow \\ &= u(f(x),x)+1 && \text{since } e = f(x) \\ &= g(x) && \text{by definition of } g \end{aligned}$$

contradiction. So $e \neq f(x)$ for any x, contradicting the assumption that f enumerates TOT (since $e \in TOT$). $\square$

EXAMPLE of an r.e. set that is not recursive

is provided by the undecidability of the Halting Problem, which in particular implies that

$$H \overset{\text{def}}{=} \{ e \in \mathbb{N} \mid \varphi_e(0) \downarrow \}$$ [cf. S_2 on p 56]

is undecidable, i.e. is not recursive. But

H <u>is</u> r.e. because $H = Dom(f)$ the domain (of definedness) of the partial recursive function

$$f(x) \overset{\text{def}}{=} u(x, 0)$$

(where u is as above)

and in general we have ...

PROPOSITION:

For a subset $S \subseteq \mathbb{N}$, the following are equivalent:

(1) S is recursively enumerable

(2) $S = \mathrm{Im}(f)$, the image of a (unary) partial recursive function

(3) $S = \mathrm{Dom}(f)$, the domain of a unary partial recursive function

(4) S is semi-decidable, meaning that the partial function

$$\mathrm{in}_S(x) = \begin{cases} 1 & \text{if } x \in S \\ \text{undefined} & \text{if } x \notin S \end{cases}$$

is partial recursive.

NOTATION :

Given a partial function $f \in Pfn(X, Y)$

$Dom(f) \overset{def}{=} \{x \in X \mid f(x)\downarrow\}$ the <u>domain</u> (of definedness) of f

$Im(f) \overset{def}{=} \{y \in Y \mid \text{for some } x \in X, f(x) = y\}$ the <u>image</u> of f

<u>Proof</u> of the Proposition

We will show $(2) \Rightarrow (1) \Rightarrow (3) \Rightarrow (4) \Rightarrow (2)$.

In all cases the implications are trivial if S is empty (since in $\emptyset$ = completely undefined function, is partial recursive and has Domain & image $= \emptyset$). So we can assume $S \neq \emptyset$, say $x_0 \in S$.

(2)⇒(1) :

Let M be a register machine computing $f(a)$ in R0 when started with R1 = a.

Construct a new machine M' computing as follows:

```
decode R1 as a pair $\langle a, t \rangle$ ;
run M for t steps starting with R1 = a and
if it halts by then, set R0 to the value it
computes in R0, else set R0 to $x_0$
```

Let f' be the unary function computed by M' (in R0, starting with input in R1).

By construction f' is total recursive and $f'(x) \in S$ for all $x \in \mathbb{N}$ (since M only computes values $f(a)$ that lie in S).

Conversely, if $y \in S = \mathrm{Im}(f)$, then $y = f(a)$ for some a. Now M computes $f(a)$ in a finite number of steps starting from $R1 = a$, say t steps. Then by construction of M' $f'(\langle a,t \rangle) = f(a) = y$. Thus every element of S is enumerated by the recursive function f' - so S is r.e.

[Remark: using the techniques of the proof of "f computable $\Rightarrow f \in PR$" one can show that S is enumerated by a primitive recursive function, since

$$f'(x) = \mathrm{ifzero}(I \dot{-} \mathrm{lab}(\mathrm{state}(\pi_1(x),\pi_2(x))), \mathrm{val}_0(\mathrm{state}(\pi_1(x),\pi_2(x))), x_0)$$

where π_1, π_2 are primitive recursive projection functions satisfying $\pi_1(\langle a,t \rangle) = a$, $\pi_2(\langle a,t \rangle) = t$, & $\langle \pi_1(x), \pi_2(x) \rangle = x$.]

(1) ⇒ (3):

Since we are assuming $S \neq \emptyset$,
$S = \{ f(n) \mid n \in \mathbb{N} \}$ for some recursive function f.

Then

$$g(x,y) \overset{\text{def}}{=} \begin{cases} 0 & \text{if } f(y) = x \\ 1 & \text{if } f(y) \neq x \end{cases}$$

is also recursive, since $g(x,y) = 1 \dot{-} eq(f(y), x)$.
Thus $\mu(g)$ is partial recursive, and

$$\begin{aligned} x \in Dom(\mu(g)) &\Leftrightarrow \mu(g)(x)\downarrow \\ &\Leftrightarrow g(x,y) = 0 \text{ for some } y \\ &\Leftrightarrow f(y) = x \text{ for some } y \\ &\Leftrightarrow x \in S \end{aligned}$$

Thus $S = Dom(\mu(g))$, as required.

(3) ⇒ (4):

If $S = Dom(f)$ with $f \in PR$, then

$$in_S(x) \equiv ifzero(f(x), 1, 1)$$

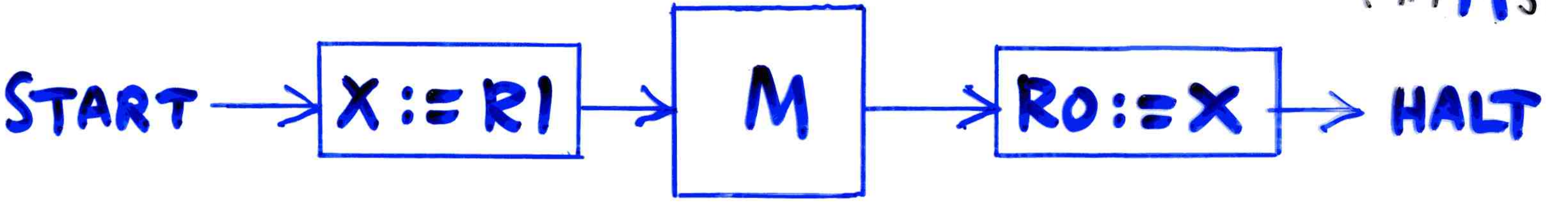
is also partial recursive, hence computable: so S is semi-decidable.

$(4) \Rightarrow (2)$:

Let M be a register machine computing $in_S(x)$ in R0 when started with x in R1.

Then

(where **X** is some register not mentioned in **M**'s program.)



computes the partial recursive function

$$f(x) \overset{\text{def}}{=} \begin{cases} x & \text{if } \; in_S(x)\downarrow \\ \uparrow & \text{if } \; in_S(x)\uparrow \end{cases}$$

and hence $\mathrm{Im}(f) = S$.

□

DEFINITION :

A subset $S \subseteq \mathbb{N}$ is called **co-r.e.** iff $\mathbb{N} \setminus S$ $\left(\overset{\text{def}}{=} \{x \in \mathbb{N} \mid x \notin S\}\right)$ is r.e.

PROPOSITION :

S is recursive if & only if it is both r.e and co-r.e.

PROOF:

$$\chi_S(x) = \begin{cases} 0 & \text{if } x \notin S \\ 1 & \text{if } x \in S \end{cases}$$

$$\chi_{\mathbb{N}\setminus S}(x) = \begin{cases} 0 & \text{if } x \in S \\ 1 & \text{if } x \notin S \end{cases}$$

$$= \mathrm{ifzero}(\chi_S(x), 1, 0)$$

So S recursive $\Rightarrow$ $\mathbb{N}\setminus S$ recursive

So S recursive $\Rightarrow$ S & $\mathbb{N}\setminus S$ both r.e.

Conversely...

Suppose

$$\left.\begin{matrix} S \\ \mathbb{N} \setminus S \end{matrix}\right\} \text{enumerated by recursive function} \left\{\begin{matrix} f \\ g \end{matrix}\right.$$

Let M be register machine which when started with x in R1 :

computes successive values of the sequence

$g(0), f(0), g(1), f(1), g(2), f(2), \ldots$

halting (at n^{th} place in sequence, say) first time get a value $= x$, and returning $\left\{\begin{matrix} 0 \\ 1 \end{matrix}\right.$ in R0 if n is $\left\{\begin{matrix} \text{even} \\ \text{odd} \end{matrix}\right.$.

Then M decides membership of S, because...

M is guaranteed to halt because f and g are total; and then

either $x \in S$ – in which case $x = f(n)$, some n

or $x \notin S$ – in which case $x = g(n)$, some n.

More formally $\chi_S(x) \equiv mod_2(\mu(h)(x))$, where

$$h(x,y) \overset{def}{=} 1 \dot{-} eq(x, ifzero(mod_2(y), g(half(y)), f(half(y))))$$

and mod_2, half, eq were defined on pages 120 & 124.

Thus χ_S is recursive because f & g are and because eq, ifzero, mod_2 & half are (primitive) recursive. □

SUMMARY

- Formalization of intuitive notion of ALGORITHM in several <u>equivalent</u> ways

  cf. "Church-Turing Thesis"

- Limitative results : undecidable problems
  uncomputable functions

  "programs as data" + diagonalization