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.

} 0, 1, 2, 3, ... stored in (idealised) registers

- jump ("goto")
- conditional ("if-then-else")

DEFINITION : a register machine consists of :

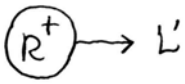
- finitely many registers R0, ..., Rn
  (each capable of storing a natural number)
- a program, consisting of a finite list of instructions of the form

  label : body of instruction

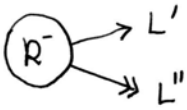
  — $(i+1)^{th}$ instruction in the list is labelled Li ($i=0,1,\ldots$)
  — instructions take one of three forms :

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 : HALT	Stop executing instructions

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 

.

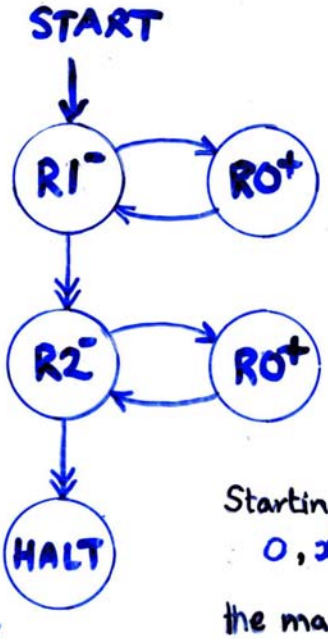
So :
- nodes of graph indicate register operations (& halting)
- arc of graph represent jumps between instructions
- labels of instructions become implicit
- loose sequential order of instructions, which is no problem so long as the START instruction of the graph is indicated.

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

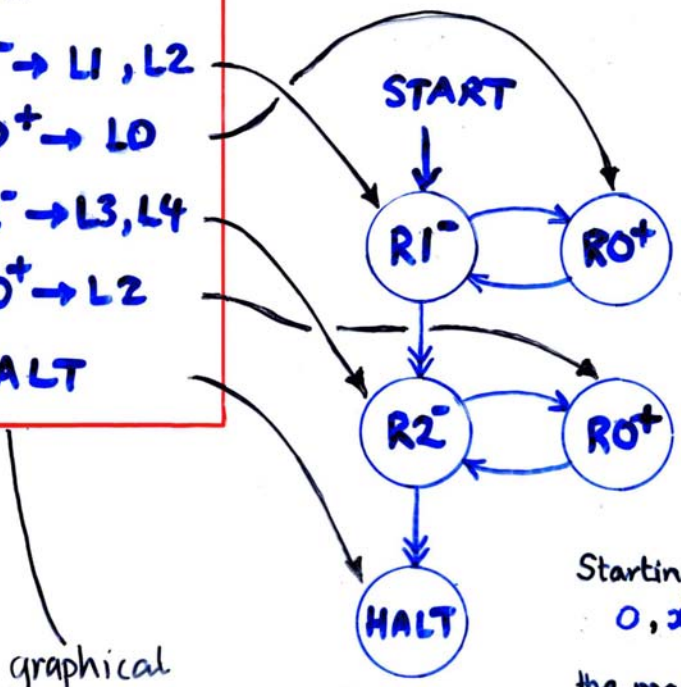
Starting with initial values 0, x, y in R0, R1, R2
the machine reaches L4 : HALT
with values x+y, 0, 0
in R0, R1, R2.

Example: register machine for addition

registers R0 R1 R2

program

L0: $R1^- \rightarrow$ L1, L2
L1: $R0^+ \rightarrow$ L0
L2: $R2^- \rightarrow$ L3, L4
L3: $R0^+ \rightarrow$ L2
L4: HALT



graphical representation

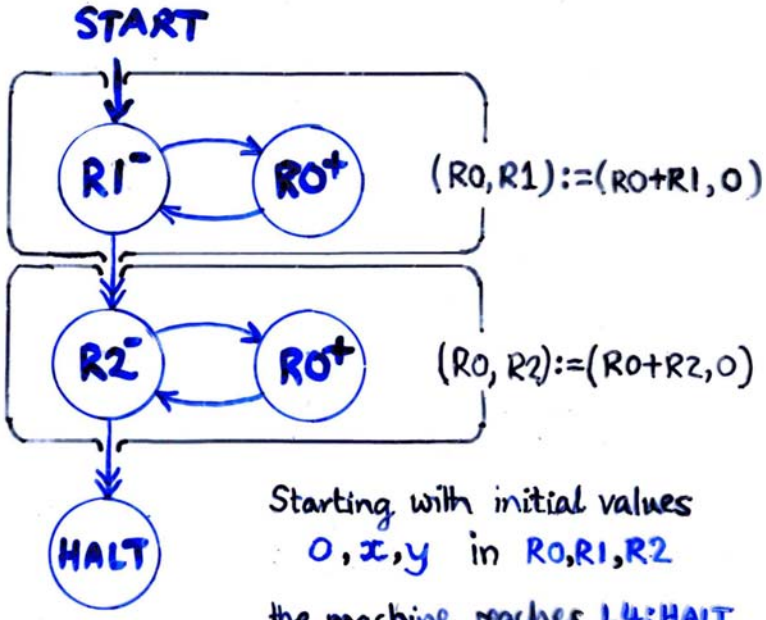
Starting with initial values 0, x, y in R0, R1, R2
the machine reaches L4: HALT
with values x+y, 0, 0
in R0, R1, R2.

Example: register machine for addition

registers R0 R1 R2

program

L0: $R1^- \rightarrow$ L1, L2
L1: $R0^+ \rightarrow$ L0
L2: $R2^- \rightarrow$ L3, L4
L3: $R0^+ \rightarrow$ 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+ → L1
L2 : 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 ("erroneous halt") ← e.g.

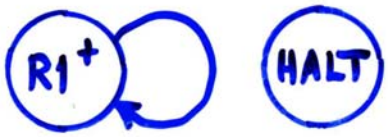
```
L1 : R1+ → L3
L2 : HALT
```

halts erroneously

Graphical representation of

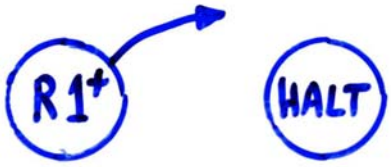
```
L1 : R1+ → L1
L2 : HALT
```

is




```
L1 : R1+ → L3
L2 : 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 $X \times Y \stackrel{\text{def}}{=} \{(x,y) \mid x \in X \;\&\; y \in Y\}$ 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 " " " "

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

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

those $f \in Pfn(X,Y)$ such that $f(x)\!\downarrow$ for all $x \in X$

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 $y \in \mathbb{N}$

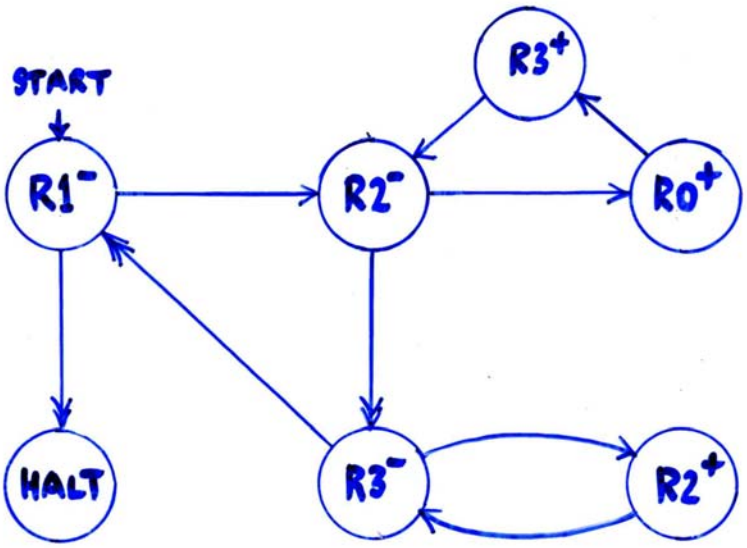
$f(x_1, \ldots, x_n) = y$ if & only if the computation of M starting with $R1 = x_1, \ldots, Rn = x_n$, and all other registers $= 0$, halts with $R0 = y$.

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

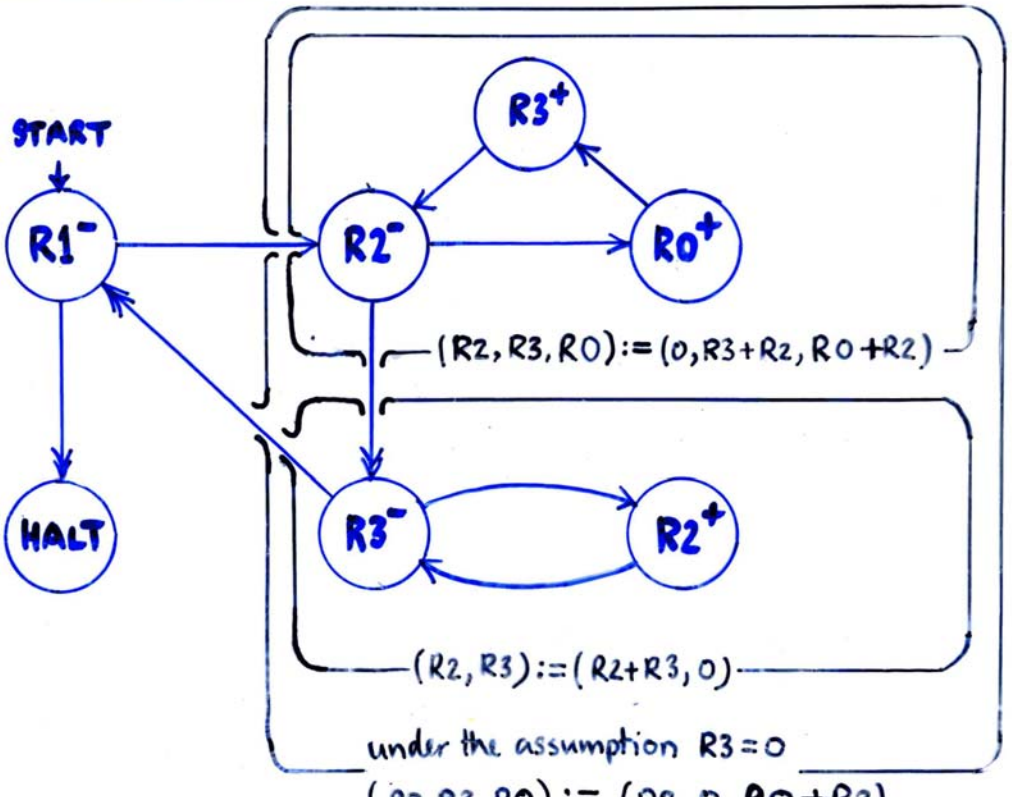
N.B. there may be many different register machines that compute the same partial function.

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 $f(x,y) \stackrel{def}{=} x$

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

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

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

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

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

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

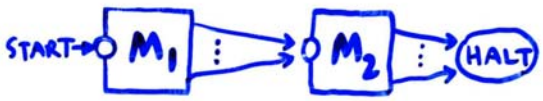
Proof – left as an exercise in register machine programming!

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.

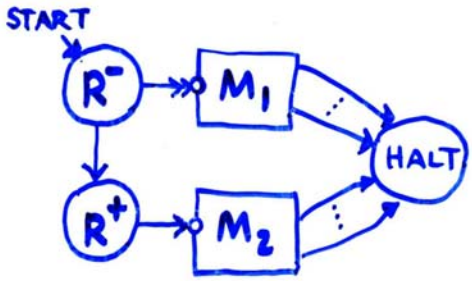
First note that there is no loss of generality (i.e. the class of computable functions remains unchanged) if we work with register machine programs with exactly one HALT instruction (because...). So graphs of programs look schematically like

Then we have...

Sequential composition $M_1 ; M_2$



IF R=0 THEN M_1 ELSE M_2



WHILE R≠0 DO M