Computability

- 2 approaches : operational x functional
- Church thesis

Foundation of Recursive Function Theory

We will try to identify such functions which are „computable" in common sense (without respect to a given computer or computational system)

To decrease an extreme amount of such functions we abstract domains and ranges to integers and we will considere functions of the form:

$$\boxed{f : \mathbb{N}^m \to \mathbb{N}^n}$$

where $\mathbb{N} = \{0, 1, 2, \dots\}$, $m, n \in \mathbb{N}$

We will call this funcions partial functions with classification:

Partial functions
- Total functions
- Strictly partial functions

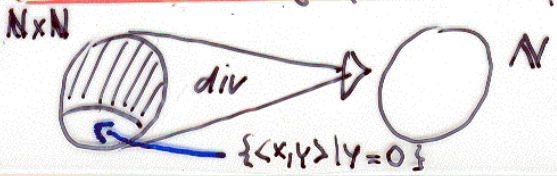
Example:

Total function

$$\boxed{\begin{array}{l} plus : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \\ plus(x, y) = x + y \end{array}}$$

Strictly part. function

$$\boxed{\begin{array}{l} div : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \\ div(x, y) = \text{integer part of } x/y, \text{ if } y \neq 0 \end{array}}$$



Convention:

a tuple $(x_1, x_2, \dots x_n) \in \mathbb{N}^n$ will be denoted as $\bar{x}$

Initial functions

The hierarchy of computable function is based on suitably chosen so called initial functions representing elementary functions (building stones for more complex functions). There are the following initial functions:

1. Zero function

$$\xi(\,) = 0$$

The function maps „empty tuple" to zero

2. Successor function

$$\sigma : \mathbb{N} \to \mathbb{N}$$
$$\sigma(x) = x + 1$$

3. Projections

$$\pi_k^n : \mathbb{N}^n \to \mathbb{N}$$

The function extracts from n-tuple the k-th component

Ex.: $\pi_2^3(7,6,4) = 6$ or $\pi_1^2(5,17) = 5$

Special case: $\pi_0^n : \mathbb{N}^n \to \mathbb{N}^0$

$$\pi_0^3(1,2,3) = (\,)$$

Primitive Recursive Functions

These functions are defined by 3 principles of joining functions:

1. Combination

The combination of two functions $f: \mathbb{N}^k \to \mathbb{N}^m$ and $g: \mathbb{N}^k \to \mathbb{N}^n$ is a function:

$$\boxed{\begin{array}{l} f \times g : \mathbb{N}^k \to \mathbb{N}^{m+n} \\ \text{such that} \quad f \times g(\bar{x}) = (f(\bar{x}), g(\bar{x})) \end{array}}$$

Ex.: $\pi_1^3 \times \pi_3^3 (4, 12, 8) = (4, 8)$

2. Composition

The composition of two functions $f: \mathbb{N}^k \to \mathbb{N}^m$ and $g: \mathbb{N}^m \to \mathbb{N}^n$ is a function

$$g \circ f : \mathbb{N}^k \to \mathbb{N}^n$$

such that $g \circ f(\bar{x}) = g(f(\bar{x}))$

Ex.:

$\sigma \circ \xi() = 1 \qquad \sigma \circ \sigma \circ \xi() = 2$

$\sigma \circ \pi_1^2(3, 1) = 4$

3. Primitive recursion

Example

Let us suppose we wanted to define a funtion $f: \mathbb{N}^2 \to \mathbb{N}$ such that $f(x,y)$ would be the number of nodes in a full, balanced tree in which every nonleaf node has x childern (x-ary tree) with depth of y.

We can see:

(1) $f(x,0) = 1$

(2) The tree of depth n has x^n leafs. If we increase the depth to $n+1$ we must add x^{n+1} nodes

$$x^{n+1} = x^n \cdot x$$

Now we are prepared to define the function f using following principle:

$$f(x,0) = x^0$$
$$f(x,y+1) = f(x,y) + x^y \cdot x$$

For example: $f(3,2) = f(3,1) + 3^1 \cdot 3 = 4 + 9 = 13$

$f(3,1) = f(3,0) + 3^0 \cdot 3 = 1 + 3 = 4$

3. <u>Primitive recursion</u> is a technique which defines a function f

$$f: \mathbb{N}^{k+1} \to \mathbb{N}^m$$

from two functions g and h

$$g: \mathbb{N}^k \to \mathbb{N}^m$$

$$h: \mathbb{N}^{k+m+1} \to \mathbb{N}^m$$

by the following prescription:

$$f(\bar{x}, 0) = g(\bar{x})$$

$$f(\bar{x}, y+1) = h(\bar{x}, y, f(\bar{x}, y)) \qquad \bar{x} \in \mathbb{N}^k$$

A schema of computation (ilustration for $y = 3$)

$$f(\bar{x}, 3) = h(\bar{x}, 2, f(\bar{x}, 2))$$

$$f(\bar{x}, 2) = h(\bar{x}, 1, f(\bar{x}, 1))$$

$$f(\bar{x}, 1) = h(\bar{x}, 0, f(\bar{x}, 0))$$

$$f(\bar{x}, 0) = g(\bar{x})$$

e.g. two „directions"

- forward (until $f(\bar{x}, 0)$ is computed)
- backward (untie computation of $f(x, y)$ is completed)

Example

Let us consider the function

$$plus : \mathbb{N}^2 \to \mathbb{N}$$
$$plus(x,y) = x+y$$

Using primitive recursion we can define the function plus in this way:

$$plus(x,0) = \pi_1^1(x)$$
$$plus(x,y+1) = \sigma \circ \pi_3^3(x,y,plus(x,y))$$

e.g.

$x+0 = 0$

$x+(y+1)$ is obtained (recursively) by finding the successor of $x+y$

<u>Definition</u>

The class of <u>primitive recursion functions</u> is a class of all function constructed from initial function using combination, composition, and primitive recursion, including initial funtions.

<u>Theorem</u>

Any primitive recursive function is a total function.

<u>Proof</u>.

(a) initial function are total functions

(b) ~~by~~ application of combination, composition or primitive recursion to total funtions results to a total funtion

Typical primitive recursive functions (PRF)

PRF's are used in typical applications of computers.

<u>Convention</u>: Instead functional notation we will used common notation frequently:

Instead $h \equiv plus \circ (\Pi_1^3 \times \Pi_3^3)$

we use $h(x,y,z) = plus(x,z)$ or

$h(x,y,z) = x+z$

Constant functions

We introduce function K_m^n which to any tuple $\bar{x} \in \mathbb{N}^n$ asigns the constant value $m \in \mathbb{N}$:

$$K_m^0 \equiv \underbrace{\sigma \circ \sigma \circ \ldots \circ \sigma}_{m\text{-times}} \{$$

Also for $n>0$ function K_m^n is PRF:

$$K_m^n(\bar{x},0) = K_m^{n-1}(\bar{x})$$

$$K_m^n(\bar{x},y+1) = \Pi_{n+1}^{n+1}(\bar{x},y,K_m^n(\bar{x},y))$$

For example: $K_3^2(1,1) = \Pi_3^3(1,0,K_3^2(1,0)) = K_3^1(1) = K_3^1(0) =$
$= K_3^0() = 3$

Constants from $\mathbb{N}^n$ can be constructed by combination of K_m^n. For example $K_2^3 \times K_5^3(x,y,z) = (2,5)$.

Multiplication

$$mult(x,0) = K_0^1(x)$$
$$mult(x,y+1) = plus(x, mult(x,y))$$

Exponentation

$$exp(x,0) = K_1^1(x)$$
$$exp(x,y+1) = mult(x, exp(x,y))$$

Predecessor funtion

$$pred(0) = f()$$
$$pred(y+1) = \Pi_1^2(y, pred(y))$$

Function monus (modification of minus)

$$monus(x,0) = x$$
$$monus(x,y+1) = pred(monus(x,y))$$

e.g. $monus(x,y) = \begin{cases} x-y & \text{if } x \geq y \\ 0 & \text{otherwise} \end{cases}$

notation : $monus(x,y) \equiv x \dot{-} y$

Function equal

$$eq(x,y) = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{if } x \neq y \end{cases}$$

$$eq(x,y) = 1 \dot{-} ((y \dot{-} x) + (x \dot{-} y))$$

$$eq(5,3) = 1 \dot{-} ((3 \dot{-} 5) + (5 \dot{-} 3)) = 1 \dot{-} (0+2) = 1 \dot{-} 2 = 0$$

We can define also

$$\neg eq \equiv monus \circ (k_1^2 \times eq),$$
$$\equiv 1 \dot{-} eq$$

Tabular function

Let us consider functions of the type

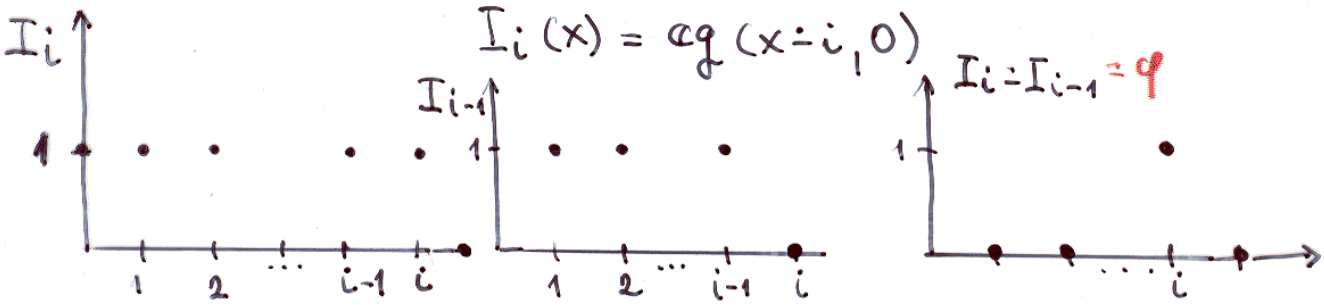
$$f(x) = \begin{cases} 3 & \text{if } x = 0 \\ 5 & \text{if } x = 4 \\ 2 & \text{otherwise} \end{cases}$$

which are usualy defined by a table. These functions can be desribed using a characteristic functions

$$\varphi_i(x) = \begin{cases} 1 & \text{if } x = i \\ 0 & \text{otherwise} \end{cases}$$

which can be expressed as

$$monus(I_i, I_{i-1}) \text{ where}$$
$$I_i(x) = eq(x \dot{-} i, 0)$$



Tabelar functions can now be created by finit sum of multiplications of constants with φ_i or $\neg\varphi_i$

Example (function f above)

$$f \equiv mult(3, \varphi_0) + mult(5, \varphi_4) + mult(2, mult(\neg\varphi_0, \neg\varphi_4))$$

Function quo (quotient)

$$quo(x,y) = \begin{cases} \text{integer part of } x/y \text{ if } y \neq 0 \\ 0 \text{ if } y = 0 \end{cases}$$

Using primitive recursion:

$$quo(0,y) = 0$$

$$quo(x+1,y) = quo(x,y) + eq(x+1, mult(quo(x,y), y) + y)$$

Beyond Primitive Recursive Function

There are functions which are computable, but which are not PRFs. Such functions are all strictly partial functions (as div) but even some total functions. An example of such total function is so called Ackermann's function (1928), which is defined by equations:

$$A(0,y) = y+1$$

$$A(x+1,0) = A(x,1)$$

$$A(x+1,y+1) = A(x, A(x+1,y))$$

e.g. $A : \mathbb{N}^2 \to \mathbb{N}$

Theorem

There is a total function from $\mathbb{N}$ to $\mathbb{N}$ which is computable but not PRF.

Proof: Considering PRFs as strings, we can ordered* them: $f_1, f_2, \dots f_n \dots$
Let us define the function $f: \mathbb{N} \to \mathbb{N}$ now such that

$$f(n) = f_n(n) + 1$$

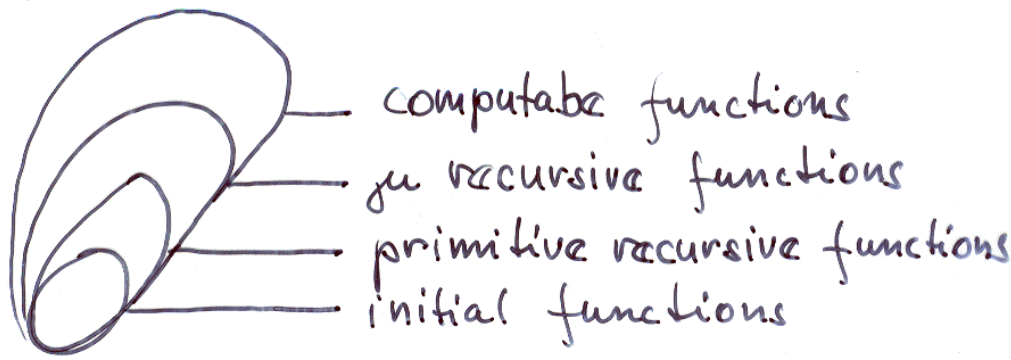
for all $n \in \mathbb{N} \setminus \{0\}$.

Clearly, f is total and computable function. But f is not PRF. Indeed if f woud be a PRF then

$$f \equiv f_m \text{ for some } m \in \mathbb{N}.$$

But in this case $f(m) = f_m(m)$ but not $f_m(m) + 1$ according to definition of.

Definition The class of total computable function is called <u>μ recursive functions</u>.



*) in lexicographical order for example

Partial recursive functions

To extend the class of computable functions beyond total computable functions we introduce a technique called minimalisation:

This technique enables to construct function

$$f: \mathbb{N}^n \to \mathbb{N}$$

from function $g: \mathbb{N}^{n+1} \to \mathbb{N}$

in such way that $f(\bar{x})$ is the smallest y for which

(1) $g(\bar{x}, y) = 0$

(2) $g(\bar{x}, z)$ is defined for all $z < y$, $z \in \mathbb{N}$

For this construction we will used notation

$$f(\bar{x}) = \mu y\, [g(\bar{x}, y) = 0]$$

Examples

1. $f(x) = \mu y\, [plus(x, y) = 0]$, e.g. $f(x) = \begin{cases} 0 & \text{if } x = 0 \\ \text{nondef} & \text{otherwise} \end{cases}$

2. $div(x, y) = \mu t\, [((x+1) \dot{-} (mult(t, y) + y)) = 0]$

3. $i(x) = \mu y\, [monus(x, y) = 0]$ – identical function

Indeed, a function defined by minimalisation is computable. The computation of $f(\bar{x})$ includes computations of

$$g(\bar{x},0)$$
$$g(\bar{x},1)$$
$$g(\bar{x},2)$$
$$\vdots$$

until (a) $g(\bar{x},y) = 0$ (then $f(\bar{x}) = y$)

or (b) $g(\bar{x},z)$ is not defined ($f(\bar{x})$ is not defined)

<u>Definition</u>

The class of partial recursive functions is the class of partial functions which can be constructet from initial functions using

(a) combination
(b) composition
(c) primitive recursion
(d) minimalisation

Turing-complutable functions

Definicion

A Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_F)$ **computes** a partial function

$$f : \Sigma^{*m} \to \Sigma^{*n} \qquad (\Sigma \subseteq \Gamma,\ \Delta \notin \Sigma)$$

iff for any $(w_1, w_2, \dots, w_m) \in \Sigma^{*m}$ and corresponding initial configuration

$$\underline{\Delta}\, w_1\, \Delta\, w_2\, \Delta \dots \Delta\, w_m\, \Delta\Delta\Delta \dots$$

the machine M

(1) stops in the case when $f(w_1, w_2, \dots, w_m)$ is defined with content of the tape

$$\underline{\Delta}\, v_1\, \Delta\, v_2\, \Delta \dots \Delta\, v_n\, \Delta\Delta\Delta$$

and $(v_1, v_2, \dots, v_n) = f(w_1, w_2, \dots, w_m)$

(2) cycles or stops abnormaly when $f(w_1, w_2, \dots, w_m)$ is not defined

The partial function which can be computed by some Turing machine is said the **Turing-computable function**.

Examples

1.

The machine computes the function $f(w_1, w_2, w_3) = (w_1, w_3)$

2. The function

$$g(w) = \begin{cases} 1 & \text{iff } w = \sigma(M) \text{ for the selfterminating machine } M \\ 0 & \text{otherwise} \end{cases}$$

is not Turing computable function.

Turing Computability of Partial Recursive Functions

Let us consider Turing machines with alfabet $\Sigma = \{0,1\}$ and partial functions $f: \mathbb{N}^m \to \mathbb{N}^n$

m-tuples (n-tuples) will be coded as tuples of binary numbers:

$\Delta 11 \Delta 10 \Delta 100 \Delta\Delta$ represents 3-tuples $(3,2,4)$

Theorem

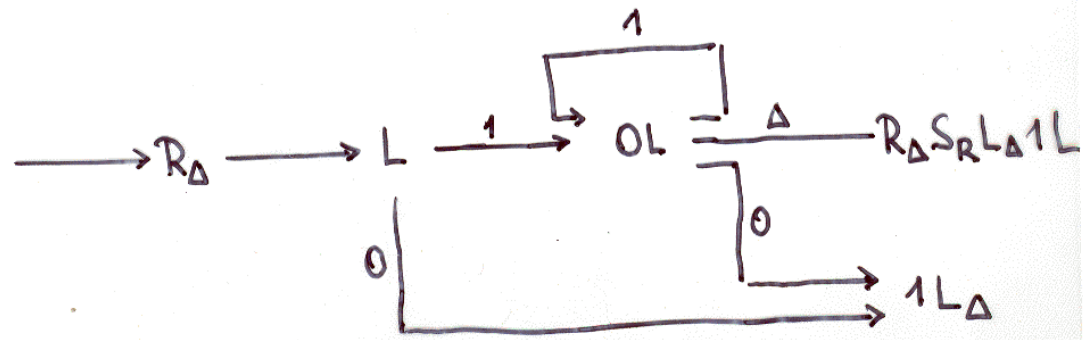
Every partial recursive function is Turing-computable.

Proof. 1. At first we have to find Turing machines which compute the initial functions ξ, σ, π:

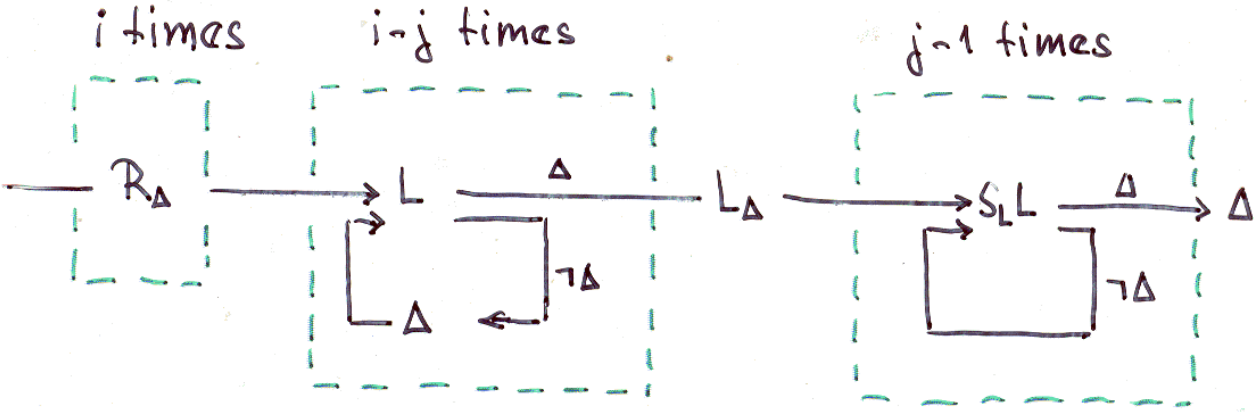
(a) ξ: $\to$ ROL

(b) a machine to compute σ

σ:



(c) a machine to compute π_j^i



2. Now we construct Turing machines for applications of (a) combination, (b) composition, (c) primitive recursion and (d) minimalization:

(a) combination

Let T. machine M_1, resp. M_2 computes partial functions g_1, resp. g_2. We construct the T. machine M that computes the function $g_1 \times g_2$:

M is 3-tape T. machine with following behaviour:

(i) M starts with coping 1st tape (input) to 2nd and 3rd tape

(ii) then M simulates the machine M_1 using 2nd tape and the machine M_2 using 3rd tape

(iii) If M_1 and M_2 terminate normaly then copies the content of 2nd and 3rd tap (separated by Δ) to 1st tape and stops.

(b) The composition $g_1 \circ g_2$ is done by the machine

$\longrightarrow M_1 M_2$

(c) primitive recursion

Considere the schema of primitive recursion

$$f(\bar{x},0) = g(x)$$
$$f(\bar{x},y+1) = h(\bar{x},y,f(\bar{x},y))$$

where p. functions g, resp. h are computed by machines M_1, resp. M_2. The function f can be computed by Turing machine M which has to do following actions:

(I) If the last tape symbol is 0, then M deletes this symbol, goes back to origin of the tape and simulates the machine M_1

(II) If tha last symbol is not 0, then the tape contens sequence:

$$\Delta\bar{x}\Delta y+1\Delta\Delta\Delta \quad \text{for some } y+1>0$$

Then (i) using machines for coping and decrementation (viz exercice) M transforms the tape to:

$$\Delta\bar{x}\Delta y\Delta\bar{x}\Delta y-1\Delta\bar{x}\Delta y-2\Delta\cdots\Delta\bar{x}\Delta 0\Delta\bar{x}\Delta\Delta$$

Then M shifts the head just beyond 0 and simulates M_1

(ii) Now the tape will be

$$\Delta\bar{x}\Delta y\Delta x\Delta y-1\Delta\ \ldots\ \Delta\bar{x}\Delta 0\Delta g(\bar{x})\Delta\Delta$$ ~~e.g.~~ i.e.

$$\sim\!\sim \quad \shortparallel \quad \sim\!\sim \quad {}_{\uparrow}\bar{x}\Delta 0\Delta f(\bar{x},0)\Delta\Delta$$

Then M shifts the head before the last $\bar{x}$ a simulates M_2

The result is $\Delta\bar{x}\Delta y\Delta x\Delta y-1\Delta\ \ldots\ \Delta x\Delta 1\Delta h(\bar{x},0,f(\bar{x},0))\Delta\Delta$

i.e. $\sim\!\sim \quad \shortparallel \quad \sim\!\sim \quad f(\bar{x},1)\Delta\Delta$

(iii) Continue with applications of M_2 to the rest of the tape. The last application of M_2 to $\Delta\bar{x}\Delta y f(\bar{x},y)$ gives $\Delta h(\bar{x},y,f(\bar{x},y))\Delta\Delta$ which is the required output $\Delta f(x,y+1)\Delta\Delta\ldots$

(d) minimalization

Let us consider a partial function

$$\mu y\,[g(\bar{x}, y) = 0]$$

where the p. function g is computed by a machine M_1.
Construct 3-tape machine M with following actions:

1. Write 0 to the tape 2
2. Copy 1st tape and 2nd tape (in sequence) to the 3rd tape
3. Simulate machine M_1 with the 3rd tape
4. If tape 3 is 0, delete the content of tape 1, copy the tape 2 to tape 1 and halt. Otherwise increment the tape 2, erase tape 3 and continue with step 2.

The partial recursive nature of Turing machines

Theorem

Any computational process performed by a Turing machine is actualy the process of computing a partial recursive function.

Proof:

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_F)$ be a Turing machine and let $b = |\Gamma|$. We will interprete the content of the tape as positive integer with b-base representation stored in reverse order. For example, if $\Gamma = \{x, y, \Delta\}$ with coding $x \approx 1$, $y \approx 2$, $\Delta \approx 0$ (always) then the content of the tape $\Delta y x \Delta\Delta y \Delta\Delta$ is equivalent 02100200... and after reversion ..00200120 is the number 501. $(b = 3)$

With this interpretation we can considere a run of T. machine M as a computation of partial function

$$f: \mathbf{N} \to \mathbf{N}.$$

which maps the numbers corresponding to initial content of the tape to the number corresponding to the content of the tape after M halts.

Let M' be coded machine M, where the codes of states are: $0 \approx q_0$, $1 \approx q_F$ and other states are coded by integers $2, 3, \ldots, k-1$, if $|Q| = k$. Because states and symbols are coded by integers, we can represent transition function δ by integer functions:

$$mov(p,x) = \begin{cases} 2 & \text{if } \delta(p,x) = (*,R) \\ 1 & \text{if } \delta(p,x) = (*,L) \\ 0 & \text{otherwise} \end{cases}$$

$$sym(p,x) = \begin{cases} Y & \text{if } \delta(p,x) = (*,Y) \\ x & \text{otherwise} \end{cases}$$

$$state(p,x) = \begin{cases} q & \text{if } \delta(p,x) = (q,*) \\ k & \text{if } p=0 \text{ or } \delta(p,x) \text{ is not defined} \end{cases}$$

The functions mov, sym and state are tabular functions and hence partial recursive functions.

Now consider a configuration of T. machine as a triple

$$(w, p, n)$$

where w is content of the tape, p is the current state, and n, $n \geq 1$, is the position of the head.

A symbol under the head can be computed from the configuration by p. recursive function cursym:

$$\boxed{\text{cursym}(w, p, n) = \text{quo}(w, b^{n-1}) \dot{-} \text{mult}(b, \text{quo}(w, b^{n}))}$$

Let it ilustrate for $w = 1120121$, $n = 5$, $b = 3$

$w = 1120121$

$\downarrow$ devide b^{n-1}

112
110 $\xrightarrow{\dot{-}}$ 2

$\uparrow$ multiple b

11

$\uparrow$ devide b^{n}

$w = 1120121$

We define anther functions:

$$\boxed{\begin{aligned}\text{nexthead}(w, p, n) = n \dot{-} \text{eq}(\text{mov}(p, \text{cursym}(w, p, n), 1) + \\ \text{eq}(\text{mov}(p, \text{cursym}(w, p, n), 2)\end{aligned}}$$

specifies next position of the head (0 value means errornous shift)

$$\boxed{\begin{aligned}\text{nextstate}(w, p, n) = \text{state}(p, \text{cursym}(w, p, n) + \\ \text{mult}(k, \neg \text{nexthead}(w, p, n))\end{aligned}}$$

(If nexthead = 0 (errornous shift) then then nextstate evaluates to value greater than $k-1$, i.e. illegal state.)

and finaly

$$\boxed{\begin{aligned} nexttape(w,p,n) = (w \dot{-} mult(b^n, cursym(w,p,n))) + \\ mult(b^n, sym(p, cursym(w,p,n))) \end{aligned}}$$

which evaluates to integer representing the new content of the tape

By combination of previous functions we construct a function step which models one step of Turing machine:

$$\boxed{step = nexttape \times nextstate \times nexthead}$$

Now we define a function $run: \mathbb{N}^4 \to \mathbb{N}^3$; $run(w,p,n,t)$ performs t transitions (steps) from configuration (w,p,n):

$$\boxed{\begin{aligned} run(w,p,n,0) &= (w,p,n) \\ run(w,p,n,t+1) &= step(run(w,p,n,t)) \end{aligned}}$$

A function computed by the machine M' (for input w) is „value" of the tape after the final state (state 0) is reached. The number of required steps is given by a function stoptime:

$$\boxed{stoptime(w) = \mu t\,[\pi_2^3(run(w,1,1,t)=0]}$$

We can conclude:

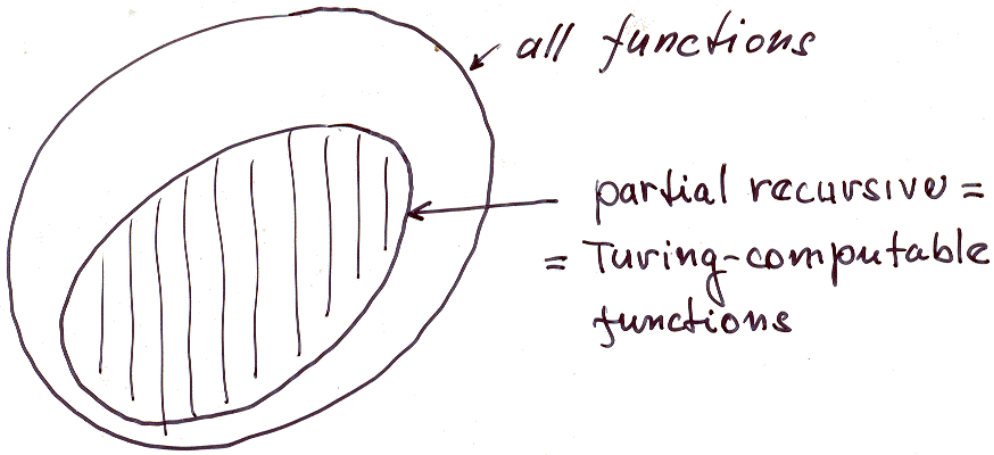
Partial function which represents the machine M' is the function $f: \mathbb{N} \to \mathbb{N}$

$$\boxed{f(w) = \pi_1^3(run(w,1,1,stoptime(w)))}$$

It is clear from the construction that f is partial recursive function.

Q.E.D.

To conclude:



The power of programming languages

A Bare-Bones Programming Language

We introduce a „minimal" programming language PL_{BB} with the following expression facilities:

Data structures: simple integer variables representing by indentifiers (non-negative values)

statements:

(a) incr name

(b) decr name

(c) while name $\neq 0$ do

⋮

end

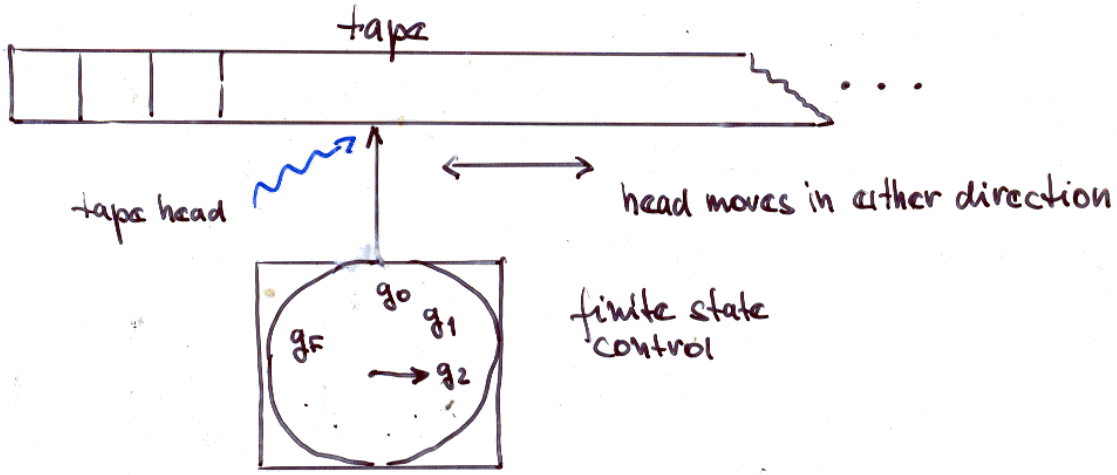
Theorem: Any partial recursive function can be programmed in PL_{BB} and any PL_{BB} program describes some partial recursive functions.

(Proof: homework)

7. TURING MACHINES

Alan M. Turing, Emil E. Post - 1936

7.1. Basic definition of Turing machine



Turing thesis:

The computational power of Turing machines is as great as any possible computational system.

Definition 7.1.

Turing machine is the 6-tuple

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_F), \text{ where}$$

(1) Q is a finite set of states

(2) Σ is a finite set of nonblanc symbols, called machine (input) alphabet

(3) Γ is a finite set of symbols called tape alphabet, $\Sigma \subseteq \Gamma$

(4) δ is mapping

$$\delta : (Q \setminus \{q_F\}) \times \Gamma \to Q \times (\Gamma \cup \{L, R\}), \quad L, R \notin \Gamma$$

(5) q_0 is initial state

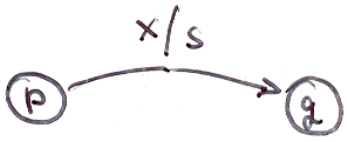
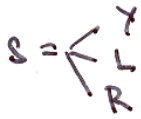
(6) q_F is final state, $q_0, q_F \in Q$

Comments :

$\Delta \in \Gamma$, Δ denotes the blank symbol

An example of notation for tape contents : $\Delta x y z \underline{z} \Delta x \Delta\Delta \ldots$

The semantics of transition function δ :

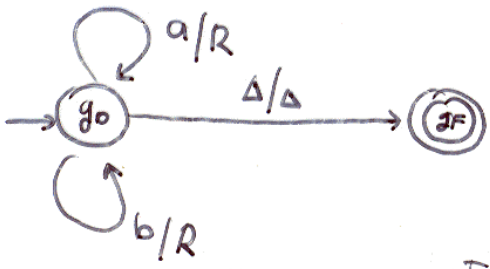
	Meaning	Graphical representation
1. $\delta(p,x) = (q,y)$ $p,q \in Q$, $x,y \in \Gamma$	write operation (replace x by y)	
2. $\delta(p,x) = (q,L)$	move tape head one cell to the left	
3. $\delta(p,x) = (q,R)$	move right	
	initial state	
	final state	

Termination of computation

(a) normal termination – moving to the final state

(b) abnormal termination
- for given (p,x) is $\delta(p,x)$ undefined
- „fall off" the left end of the tape

Example



T. machine that shifts the head to the first symbol Δ right from the current position

For example :

$\Delta \underline{u} a b \Delta b u \Delta \ldots \longrightarrow \Delta u a b \underline{\Delta} b a \Delta \ldots$

4.2. Modular construction of Turing machines

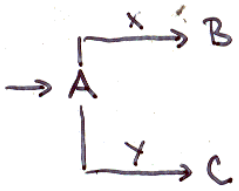
Combining T. machines

An approach is similar to the case of combining finite automata (your project).

Example 7.1.

See Fig. 3.3.

The composite machine shown in Fig. 3.3 could be summarized by the composite diagram



where
A represents machine that moves its tape head one cell to the right
B — " — that searches for an x
C — " — a y

Conventions for drawing composite diagrams:

1. The sequence of machines $\rightarrow A \rightarrow B \rightarrow C$
   could be represented as $\rightarrow ABC$

2. Parametrization

   $\xrightarrow{x,y,z} \} \xrightarrow{\omega}$ ω has value of that x, y, z which is current symbol on the tape

E.g.

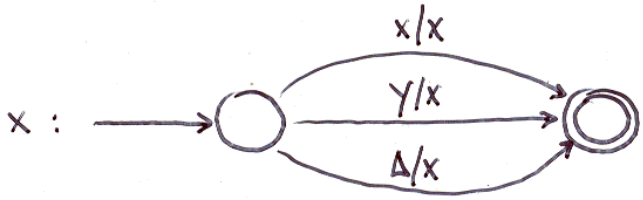
3. Branching

(The difference between $M_1 \to M_2$ and $M_1 \xrightarrow{x} M_2$)

Basic buildings blocks

Let $\Gamma = \{x, y, \Delta\}$

Machines L, R, x



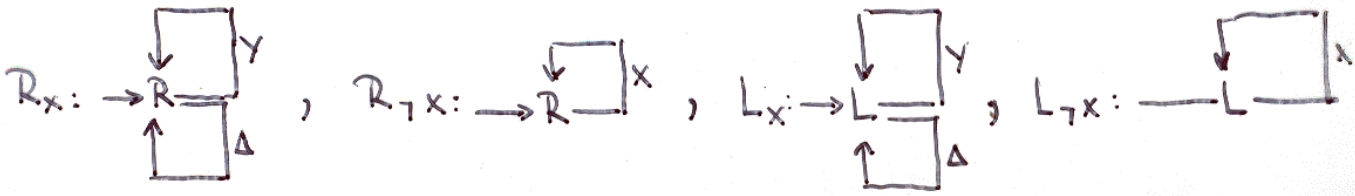
An example

$\to R \to y \to L$ or shortly $\to RyL$

transforms the tape

$\Delta xyxy\underline{x}\Delta\Delta\ldots$ to $\Delta xyxy\underline{x}y\Delta\ldots$

Machines $L_x, L_{\neg x}, R_x, R_{\neg x}$



Turing machines S_R and S_L for shifts of tape contents

The machine S_R shifts the string of non blank symbols found at the left of the current cell one cell to the right.

Examples of activations of S_R

$\Delta xyy\underline{x}x\Delta\Delta\ldots \quad \rightarrow \quad \Delta\Delta xy\underline{y}x\Delta\Delta\ldots$

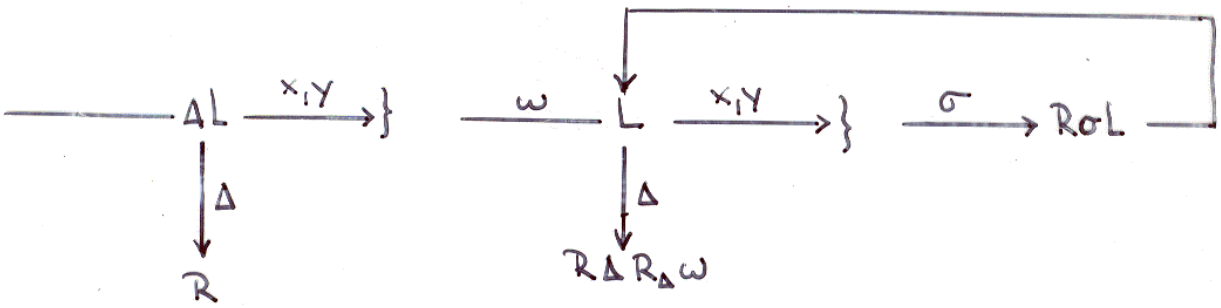
$\Delta yxy\underline{\Delta}\Delta xxy\Delta\ldots \quad \rightarrow \quad \Delta\Delta yx\underline{y}\Delta xxy\Delta\ldots$

$xy\Delta\underline{y}x\Delta\Delta\ldots \quad \rightarrow \quad xy\Delta\underline{\Delta}x\Delta\Delta\ldots$ - special case

The machine S_L works symmetricaly:

$\Delta\underline{y}yxx\Delta\Delta\ldots \quad \rightarrow \quad \Delta\underline{y}xx\Delta\Delta\ldots$

Realization of S_R:



Realization of S_L:

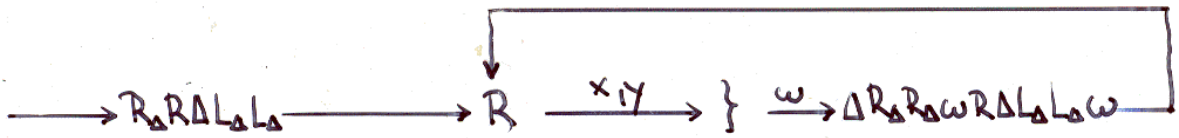
Exercise

Examples of application

(a) A copying machine

The machine transforms the tape

$\underline{\Delta} w \Delta$ to $\Delta w \underline{\Delta} w \Delta$. $w \in \{x, y\}^*$

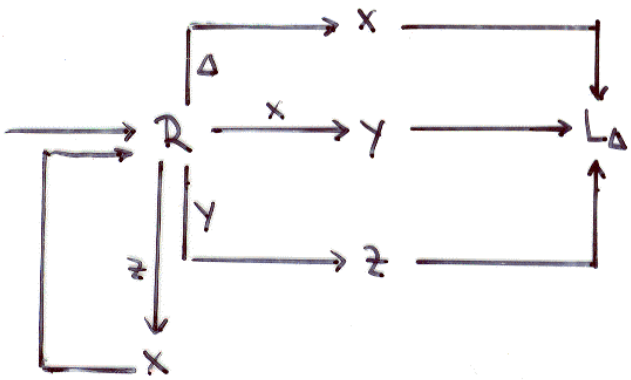


Example

1. $\underline{\Delta} x y \Delta$ $(R_\Delta R \Delta L_\Delta L_\Delta)$
2. $\underline{\Delta} x y \Delta\Delta$ $(R$
3. $\Delta \underline{x} y \Delta\Delta$ $(\omega = x, \Delta)$
4. $\Delta\underline{\Delta} y \Delta\Delta$ $(R_\Delta R_\Delta)$
5. $\Delta\Delta y \Delta \underline{\Delta}$ $(\omega R \Delta$
6. $\Delta\Delta y \Delta x \underline{\Delta}$ $(L_\Delta L_\Delta \omega)$
7. $\Delta \underline{x} y \Delta x \Delta$

⋮

(b) Strings generation



(*)

The machine produces the next strings in the sequence:

$\alpha, x, y, z, xx, yx, zx, xy, yy, zy, xz, yz, zz, xxx, yxx, \ldots$

7.3. Turing machines as languages acceptors

Definition 7.2.

A string $w \in \Sigma^*$ is said to be accepted by T. machine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_F)$ iff M starting from initial contents of the tape $\underline{\Delta} w \Delta\Delta \ldots$ reads w and terminates in q_F.

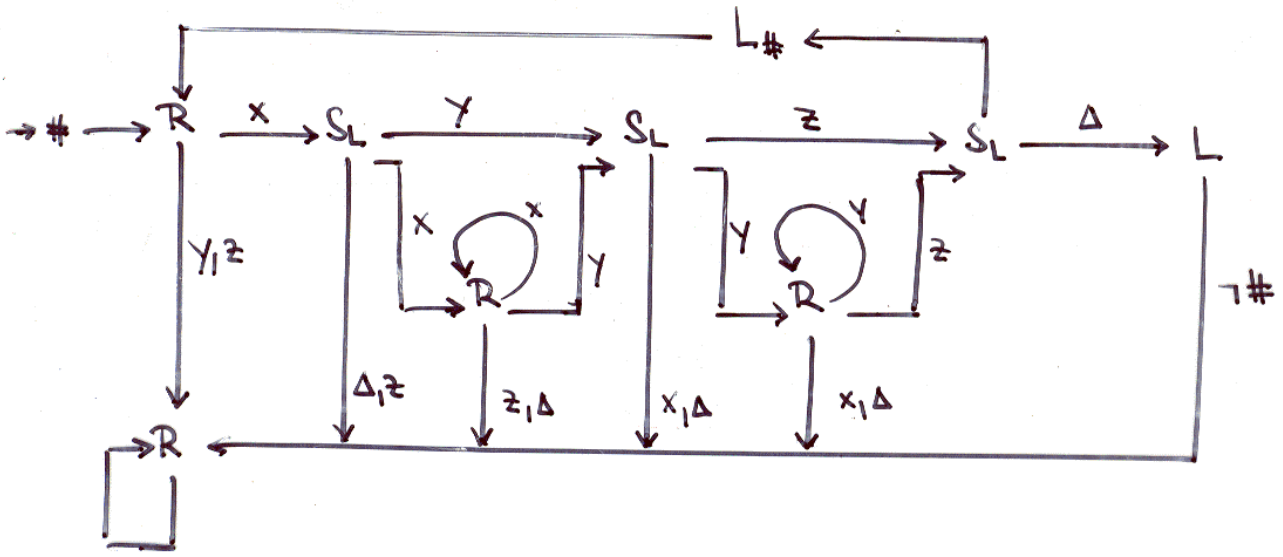
The set

$$L(M) = \{ w \mid w \text{ is accepted by } M \}$$

is said to be the language accepted by Turing machine M

Example 7.3.

Let T be T. machine



Then $L(T) = \{ x^n y^n z^n \mid n \geq 1 \}$

T repeats reductions

$$x^n y^n z^n \to x^{n-1} y^{n-1} z^n \to x^{n-1} y^{n-1} z^{n-1}$$

until ϵ (empty string) is reached

$\# \in \Gamma$ denotes the left end of the tape

An alternative way of the string acceptation by T. machine:

$$\underline{\Delta} w \Delta\Delta \rightarrow \underline{\Delta} Y \Delta\Delta \quad . \quad Y \in \Gamma \ (\text{Yes})$$

Realization

$$\longrightarrow R_\Delta S_R R * L_\Delta L \# R \longrightarrow M_0 \rightarrow R_* \rightarrow \Delta L \ \neg\# \rightarrow \Delta R Y L$$

M_0 is T. machine accepting L by going to the final state

7.4. Variants of Turing machines

7.4.1 Multiple-tape Turing machine

We will consider T. machine with k tapes and k corresponding heads. The transition function δ can be defined as follows:

$$\delta : (Q \setminus \{q_F\}) \times (\Gamma_1 \times \Gamma_2 \times \dots \times \Gamma_k) \rightarrow Q \times (\Gamma_i \cup \{L, R\}) \times \{1, 2, \dots, k\}$$

Theorem 7.1

For each k-tape Turing machine M there is a one tape Turing machine M' such that $L(M) = L(M')$.

Proof:

7.4.2. Nondeterministic Turing machines

Definition 7.3.

A Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_F)$ is said to be a nondeterministic Turing machine if δ, the transition function, has the form

$$\boxed{\delta : (Q \setminus \{q_F\}) \times \Gamma \to 2^{Q \times (\Gamma \cup \{L, R\})}}$$

Theorem 7.2.

For each nondeterministic Turing machine M there is a deterministic Turing machine M' such that $L(M) = L(M')$.

Proof NTM M will be simulated by 3-tape deterministic Turing machine M'

Purpous of the tapes:

Tape 1. input tape - it contents an input string
2. working tape
3. stories coded sequence of transitions

The activities of M' proceed as follows:

1. Copy the input string from tape 1 to tape 2
3. Generate the next transition sequence on tape 3
3. Simulate this sequence using tape 2
4. If this simulation leads to a final state in M, halt. Otherwise erase tape 2 and return to step 1

7.5 Turing machines languages and languages of type 0

(also: frase structured languages or recursivly enumerable languages)

We can define a configuration of T. machine. It is given

(1) by a state of finite control

(2) by contents of the tape

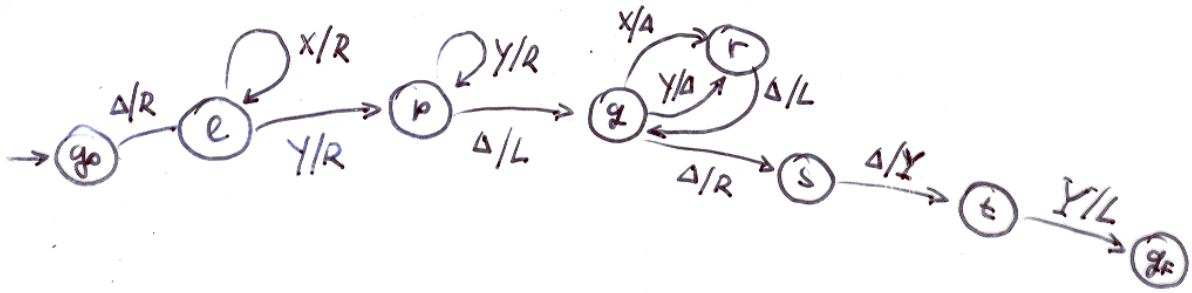
(3) by position of the head

Convention:

If current state is p, $p \in Q$ and contents of tape is $\Delta x y x \Delta$ we will represent the configuration in the form $[\Delta x p y x \Delta]$ or $[\Delta x p y x \Delta\Delta]$ or ...

Example 7.4

Let us consider the skeleton of Turing machine which accepts the language $\{x^m y^n \mid m \geq 0, n > 0.\}$



The sequence of configurations for accepting xxy:

1. $[q_0 \Delta x x y \Delta]$
2. $[\Delta \ell x x y \Delta]$
3. $[\Delta x \ell x y \Delta]$
4. $[\Delta x x \ell y \Delta]$
5. $[\Delta x x y p \Delta]$
6. $[\Delta x x q y \Delta]$
7. $[\Delta x x r \Delta]$
8. $[\Delta x q x \Delta]$
9. $[\Delta x r \Delta]$
10. $[\Delta q x \Delta]$
10. $[\Delta r \Delta]$
11. $[q \Delta]$
12. $[\Delta s \Delta]$
13. $[\Delta t Y \Delta]$
14. $[q_F \Delta Y \Delta]$

Theorem 4.3.

Each language accepted by Turing machine M is the language of type 0.

Proof. We will construct a grammar $G = (N, \Sigma, P, S)$ in which there is a derivation for each string accepted by Turing machine such that it corresponds to the reversal of the sequence of configurations accepting the string:

(1) $N = \{S\} \cup \{[,]\} \cup Q \cup (\Gamma \setminus \Sigma)$

(2) P contents productions of the form:

(a) $S \to [q_F \Delta Y \Delta]$

(b) $\Delta] \to \Delta\Delta]$ (using this production we can extend the string $[q_F \Delta Y \Delta]$ to required lenght $[q_F \Delta Y \Delta\Delta \cdots \Delta]$

(c) $qy \to px$ if $\delta(p,x) = (q,y)$

(d) $xq \to px$ if $\delta(p,x) = (q,R)$

(e) $qyx \to ypx$ for each y if $\delta(p,x) = (q,L)$

(f) $\left.\begin{array}{l} [q_0 \Delta \to e \\ \Delta\Delta] \to \Delta] \\ \Delta] \to e \end{array}\right\}$ eliminate symbols $q_0, [,], \Delta$

It can be shown that $L(G) = L(M)$ i.e.

$w \in L(M) \iff$ there is the derivation

$$S \Rightarrow [q_F \Delta Y \Delta] \Rightarrow \cdots \Rightarrow [q_0 \Delta w \Delta] \Rightarrow w\Delta] \Rightarrow w$$

Theorem 4.4.

Each language of type 0 is the language accepted by Turing machine.

Proof.

Let $L = L(G)$ be the language generated by the G of type 0. We will construct nondeterministic 2-tape Turing machine N such, that $L(G) = L(N)$.

Basic action: simulation of application of a production $\alpha \to \beta$

Contents of tapes:

1. tape – input string
2. tape – working tape for construction a derivation using substitution according productions $\alpha \to \beta$

4.6. The scope of the class of 0 type languages

Theorem 7.5.

For each alphabet Σ there is a language L over Σ such that L is not the language of type 0.

Proof.

1. At first we have to understand that any 0 type language over alphabet Σ can be accepted by T. machine with tape alphabet $\Sigma \cup \{\Delta\}$.

Really, if T. machine M works with more symbols than we can coded other tape symbols using symbol from Σ.

2. With this assumption we can now list all T. machines with tape symbols in $\Gamma = (\Sigma \cup \{\Delta\})$. We start with all 2-state machines, then 3-state, 4-state and so on.
This list can be indexed a hence the set of all such T. machines, i.e. the set of all 0-typy languages over Σ is countable.

3. On the other hand we now show, that the set 2^{Σ^*} which represents the set of all languages over Σ is uncountable. We use the schema of proof called diagonalisation. (It was used by Cantor to prove that the set $\mathbb{R}$ is uncountable):

Let us suppose that the set of all languages over Σ is countabe. Then, by definition, there is a bijection f

$$f: \mathbb{N} \leftrightarrow 2^{\Sigma^*}$$

Let's tray express this bijection in the form of infinite matrix (for $\Sigma = \{x, y, z\}$)

	ε	x	y	z	xx	xy	xz	$\cdots$	xxx	xxy	$\cdots$
$f(1)$	a_{11}	a_{12}	a_{13}		$\cdots$				a_{1i}	a_{1i+1}	$\cdots$
$f(2)$	a_{21}	a_{22}	a_{23}		$\cdots$				a_{2i}	a_{2i+1}	$\cdots$
$f(3)$	a_{31}	a_{32}	a_{33}		$\cdots$						
$\vdots$					$\ddots$						
$f(k)$	a_{k1}	a_{k2}				a_{kk}					
$\vdots$							$\ddots$				

The language $L_n = f^{\nearrow}(n)$ is coded by (infinite) binary vector:

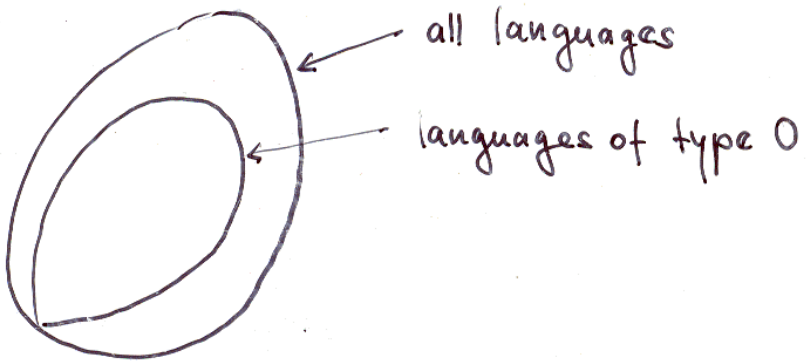
$(a_{n1}, a_{n2}, a_{n3}, \ldots\ldots)$, where $a_{nj} = \begin{cases} 0 & \text{if the } j\text{-th string is not in } L_n \\ 1 & \text{if the } j\text{-th string is in } L_n \end{cases}$

But now consider the language corresponding to the „negation" of diagonal elements:

$$\tilde{L} = (\bar{a}_1, \bar{a}_2, \bar{a}_3, \dots) \text{ where } \bar{a}_i = \begin{cases} 0 & \text{if } a_{ii} = 1 \\ 1 & \text{if } a_{ii} = 0 \end{cases}$$

The language $\tilde{L}$ differs from any languages $f(n)$, $n \in \mathbb{N}$, but $\tilde{L} \in 2^{\Sigma^*}$. It means that f is not onto mapping (surjective map.) and hence there is not bijection like f and the cardinality of 2^{Σ^*} is greater than cardinality of $\mathbb{N}$ - the set of all languages over Σ is uncountable.

Conclusions



Consequences of Turing's thesis:

1. The languages without grammatical foundation can not be recognized by any algorithmic process
2. If natural intelligence is based on the level more powerful than algorithm execution, then Turing's thesis suggests that the dream of developing truly intelligent computing machines is destinated to remain a dream forever.

4.4. Beyond phrase-structured languages

4.4.1 A coding system for Turing machine

The system includes a coding of
(1) states
(2) symbols in Γ and symbols L and R
(3) transition function δ

The result will be the binary string which represents completly given Turing machine.

state encoding:

Let Q is ordered : $Q = \{q_0, q_F, q, p, r, \dots\}$

$\boxed{\text{the code for the } j\text{-th state is } 0^j}$

e.g. $q_0 \approx 0$, $q_F \approx 00$, 5th state ≈ 00000

symbol encoding (we suppose the tape symbol in $\Sigma \cup \{\Delta\}$)

Again we order the alphabet $\Sigma = \{a_1, a_2, \dots a_n\}$ and choose the following encoding:

$\Delta \approx \epsilon$ (empty string)

$L \approx 0$

$R \approx 00$

$a_i \approx 0^{i+2}$ $\quad i = 1, 2, \dots, n$

transition encoding

the transition $\delta(p,x) = (q,y)$, $y = \begin{cases} x \in \Gamma \\ L \\ R \end{cases}$

is encoded as 4-tuple
(p,x,q,y) by concatenating the codes for p,x,q,y
and using 1 as a separator:

$$\boxed{\text{„code } p\text{"}\ 1\ \text{„code } x\text{"}\ 1\ \text{„code } q\text{"}\ 1\ \text{„code } y\text{"}}$$

Examples 7.5

The transition $\delta(q_0, x) = (q_F, R)$, the code for x is 000 , will be coded as 01000100100

$$\delta(q_0, \Delta) = (q_F, \Delta) \quad \approx \quad 011001$$

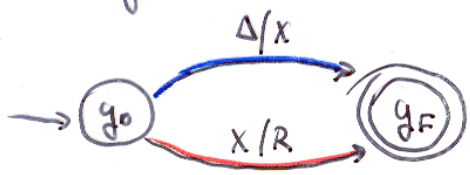
Machine encoding

concatination of all transitions (in systematic way) and separators 1 :

1 „code for 1st transition" 1 „code for 2nd transition" 1 . .
. . . 1 „code for m-th transition" 1

Example 7.6.

Turing machine



It's code

1011001000101000100100 1

An example of non phrase structural languages

The coding system enables mapping of each T. machine to the binary number (not in oposite direction).

Now let us considere the class $\mathcal{T}$ of all Turing machines over an input alphabet Σ and tape alphabet $\Sigma \cup \{\Delta\}$ and the mapping

$$\tau : \mathbb{N} \to \mathcal{T}$$

such that

$$\tau(n) = \begin{cases} T & \text{iff } n \text{ is the code of } T \\ \overline{T} & \text{otherwise} \end{cases}$$

, $\overline{T}$ is the trivial T machine:

τ is clearly onto mapping.

Now construct another function

$$\varrho : \Sigma^* \to \mathcal{T} \text{ such that}$$

for any $w \in \Sigma^*$ $\quad \varrho(w) = \tau(|w|) \equiv M_w$

ϱ is again onto mapping.

Because $w \in \Sigma^*$ and Σ is the tape alphabet of M_w, there is possible to apply M_w on string w. Let's define language L_0:

$$L_0 = \{ w \mid M_w \text{ does not accept } w \}$$

We will show that L_0 can not be accepted by any T. machine. We will suppose an opposite, i.e. there is $T_0 \in \mathcal{T}$ such that $L_0 = L(T_0)$. Then ofcause exists $w_0 \in \Sigma^*$ such that $L_0 = L(M_{w_0})$. ($T_0 \equiv M_{w_0}$) Now we are asking if $w_0 \in L(M_{w_0})$. Either $w_0 \in L(M_{w_0})$ or $w_0 \notin L(M_{w_0})$. (Tertia non datur). From definition of L_0: $w_0 \in L(M_{w_0}) \Rightarrow w_0 \notin L_0$ and $w_0 \notin L(M_{w_0}) \Rightarrow w_0 \in L_0$, since $L_0 = L(M_{w_0})$ the both implication are contradictory and hence $L_0 \notin \mathcal{L}_0$.

7.7.2 Universal Turing machines

It is a concept of „programable" machine enabling in the form of input string to specify concrete Turing machine as well as data for it.

We use the binary coding for the specification of a T. machine. In the same manner we will code also input string representing „data":

$$\underbrace{10000}\,1\,\underbrace{000}\,1\,\underbrace{00000}\,1$$

symbol codes

The whole „problem" can be now given by concatenation of machine code and data code.

For example:

$$\underbrace{\underbrace{1011001000\,1}_{\text{1st transition}}\;\underbrace{01000100\,1}_{\text{2nd transition}}}_{\text{the code of machine}}\;\underbrace{1\,0001\,0001\,000\,1}_{\text{the code of input string}}$$

The universal Turing machine which can process this specification can be 3-tape T. machine:

1st tape – programm + data (+ output data)
2nd tape – working tape
3rd tape holds a representation of current state of the machine being simulated

As we know we can construct an equivalent 1 tape machine to this one. Let us denote that machine by T_U.

7.8. Acceptable versus Decidable languages

The ability to accept a language is not symmetric to the ability to accept its complement for T. machines, i.e. Turing languages are not closed under complement. We show it using universal T. machine T_U and the language L_0.

We construct T. machine M'_w which for any string $w \in \Sigma^*$

1. computes binary number $\tau^{-1}(\rho(w))$, i.e. the code of M_w
2. concatenates the code of M_w with the code of w

Then we do composition of T. machines $\rightarrow M'_w \rightarrow T_U$

The resulting machine accepts languages L_1

$$L_1 = \{ w \mid M_w \text{ accepts the string } w \}$$

which is clearly the complement of L_0.

We see that generaly is not possible to construct T. machine M which writes „Y" if input string $w \in L(M)$ and „N" if $w \notin L(M)$.

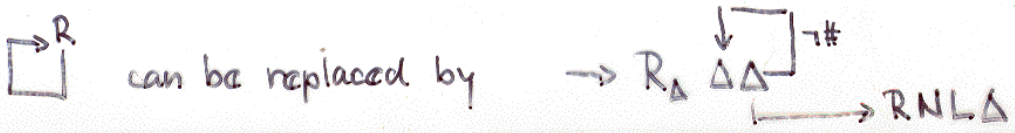
The languages for which it is not possible are called the Turing acceptable languages (or partialy decidable languages).

The language for which such T. machine exists are called Turing decidable languages.

Examples 7.7.

1. L_1 is acceptable but not decidable
2. CFL's are decidable languages
3. $L = \{ a^n b^n c^n \mid n \in \mathbb{N} \}$ is decidable

In given T. machine for L the case



Terminological note.

The class of languages accepted by Turing machine is called recursively enumerable languages

~ " ~ decidable by Turing machines is called recursive languages

7.9. The halting problem

Now we are going to prove the clasical and representative undecidable problem for Turing machine which is called the halting problem.

Let us suppose the binary alphabets of a Turing machines i.e. $\Sigma = \{0,1\}$, $\Gamma = \{0,1,\Delta\}$ and us denote the encoded version of M as $\sigma(M)$. $\sigma(M)$ is binary string which can be putted on input of M. We are interesting if the machine M for $\sigma(M)$ will be halted (M is selfterminating) or not (M is nonselfterminating).

We define the language L_F over alphabet $\Sigma = \{0,1\}$ as

$$\boxed{L_F = \{\sigma(M) \mid M \text{ is selfterminating}\}}$$

The halting problem is stated as a problem of decibility of the language L_F. This problem is, unfortunately' undecidable. We can prove it, analogicly to decibiliby of L_0, by contradiction.

So, let us suppose that there is a Turing machine, denote's it M_F, which decides the language L_F. Now let us modify the machine M_F in such way, that modyfied machine M_F' outputs 0, resp. 1 iff M_F gives a message N, resp. Y. The machine M_F' has hence tape alphabet $\{0, 1, \Delta\}$ and can be used to construction of T. machine M_0

$$M_0: \quad \longrightarrow M_F' \longrightarrow R \xrightarrow{1} R$$

We will examine if the machine M_0 is selfterminating. If L_F is decidable, then the anwer is either „yes" or „no".

(a) Let us suppose that M_0 is selfterminating. Then for $\sigma(M_0)$ the machine M_F' halts with output 1 and M_0 goes along the arc $R \xrightarrow{1} R$ and cycles forever.

Hence

M_0 is „selfterminating" => M_0 is „nonselfterminating"

(b) Let us suppose that M_0 is nonselfterminating. Then for input $\sigma(M_0)$ the machine M_F' halts with output 0 and the machine M_0 halts. Hence

M_0 is nonselfterminating => M_0 is selfterminating

From inconsistence of the both implications we conclude that the asumption of decidability of L_F was false. So the halting problem is undecidable.

7.10. Post's correspondence problem

Definition 7.

The set S of ordered pairs of strings over alphabet Σ

$$S = \{(\alpha_1, \beta_1), (\alpha_2, \beta_2), \dots, (\alpha_k, \beta_k)\}, k \geq 1, \alpha_i, \beta_i \in \Sigma^+$$

is called Post system over alphabet Σ.

The solution of Post's system is each nonempty sequence of natural numbers $i_1, i_2, \dots i_m$, $i_j \leq k$, such that

$$\alpha_{i_1} \alpha_{i_2} \dots \alpha_{i_m} = \beta_{i_1} \beta_{i_2} \dots \beta_{i_m}$$

The Post's correspondence problem:

Is there a solution for given (any) Post's system?

Theorem 7.

Post's correspondence problem is undecidable.

Proof: It can be shown that Post's correspondence problem is equivalent to the halting problem.

Example 7.7.

1. Let $S = \{(b, bbb), (babbb, ba), (ba, a)\}$ be the Post system over $\{a, b\}$.

The system has a solution $i_1 = 2, i_2 = 1, i_3 = 1, i_4 = 3$ (so $m = 4$)

$$\underbrace{babbb}_{\alpha_2}\,\underbrace{b}_{\alpha_1}\,\underbrace{b}_{\alpha_1}\,\underbrace{ba}_{\alpha_3} = \underbrace{ba}_{\beta_2}\,\underbrace{bbb}_{\beta_1}\,\underbrace{bbb}_{\beta_1}\,\underbrace{a}_{\beta_3}$$

2. On the other hand the Post system

$$S = \{(ab, abb), (a, ba), (b, bb)\}$$

has no solution, because $|\alpha_i| < |\beta_i|$ for $i = 1, 2, 3$.

The Post correspondence problem is often used for a proof of undecibility of some important problems:

1. The problem of equivalence of context-free grammars
2. The problem of non-emptiness of a language generated by context-sensitive grammar
3. The problem of membership of w in $L(G)$ where G is grammar of typy 0

8. COMPLEXITY

The study of computational processes has led to the classification of problems into two broad categories:

- the solvable problems
- the unsolvable problems

Now we will deal with the solvable problems. We concentrate our attention to questions „how problems are difficult to solve with respect of required time or space". This aspect is called complexity of problems

Two basic types of complexity are distinguished:

- time complexity
- space complexity

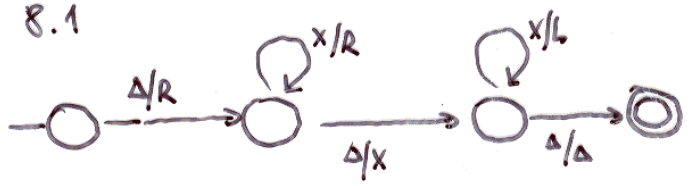
8.1. Complexity of Turing Machine Computations

Time complexity - the number of steps executed during computation

Space complexity - the number of type cells required by the computation

Example 8.1

T:



for input ΔxxxΔΔ : time complexity of computation is 9

space — " — is 5

Simple implication:

If the time complexity of computation of T is n, then space complexity of computation of T is not greater than n+1.

8.2. Complexity of algorithms and problems

The theoretical approach is based on Turing thesis :

Each algorithm is implementable by some Turing machine

Then the complexity of algorithm is done by analysis of the corresponding Turing machine. The aim is to express (quintified) required resources (time and space) as a funtion of the lenght of input strings (input data)

Usualy we distinguish

(1) worst-case analysis
(2) best-case analysis
(3) average-case analysis

Complexity of problems :

The complexity of problem is the complexity of the best (known or proved) algorithm of its solution

Example :
Find the complexity of string comparision problem

Alternative approach :

Rates (orders) of Growth :

$O(f(n))$ – asymptotical upper bound
$\Theta(f(n))$ – asymptotical tight bound
$\Omega(f(n))$ – asymptotical lower bound

8.3. Time complexity of language recognition problems

Definition 8.1.

Let M be (single or multitate) Turing machine which computes the partial function $f: \Sigma_1^* \to \Sigma_2^*$. We say that M computes the funtion in polynomial time if there is a polynomial $p(x)$ such that for each $w \in \Sigma_1^*$ for which $f(w)$ is defined, M computes $f(w)$ in no more than $p(|w|)$ steps. The funtion f is called the function computable in polynomial time.

Theorem 8.1.

Let f_1 and f_2 are the functions computable in polynomial time. Then composition $f_1 \circ f_2$ is also computable in polynomial time

Proof:

There is a Turing machine M_1 that computes $f_1(v)$ in no more than $p_1(|v|)$ steps and a Turing machine M_2 that computes $f_2(w)$ in no more than $p_2(|w|)$ steps; $p_1(x)$ a $p_2(x)$ are polynomials. The Turing machine $\to M_1 M_2$ which computes $f_2 \circ f_1(v)$ has for any v the following complexity:

$$p_1(|v|) + p_2(p_1(|v|)+1), \text{ where}$$

$p_1(|v|)$ is the complexity of computation $f_1(v)$

$p_1(|v|)+1$ is maximal length of output of M_2

$p_2(p_1(|v|)+1)$ is the complexity of computation of f_2 aplied on $f_1(v)$

Resulting complexity is doviosly polynomial.

Theorem 8.2.

The class of funtions which are computable by multi-tape Turing machines in polynomial time is the same as the class of funtions computable by single-tape T. machines in polynomial time.

Proof:

Definition 8.2.

We say that Turing machine M accepts the language L in polynomial time if $L = L(M)$ and there is a polynomial $p(n)$ such that the number of steps required to accept any $w \in L(M)$ is no greater than $p(|w|)$. We define P to be the class of languages that can be accepted by Turing machines in polynomial time.

Theorem 8.3.

If a T. machine M accepts the language L in polynomial time, then there is another T. machine M' that also accepts L in polynomial time but indicates acceptance by halting with tape configuration $\underline{\Delta}\,Y\,\Delta\,\Delta\ldots$

Proof:

Theorem 8.4.

Let L be a language accepted by multi-tape Turing machine in polynomial time. Then L belongs to class P.

Proof: We define the function f

$$f(w) = \begin{cases} Y & \text{if } w \in L \\ \text{is not defined} & \text{if } w \notin L \end{cases}$$

and apply Theorem 8.2.

Conclusion: Robustness of the class P

8.4. Polynomial-time decidable languages

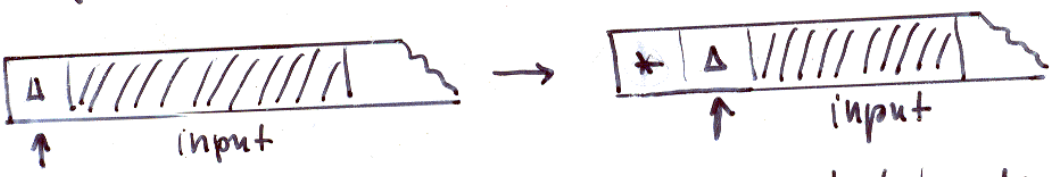
Definition of polynomial-time decidable languages is obvious.

Q: What is relation between polyn.-time acceptable and polyn.-time decidable languages?

Theorem 8.5.

If L is polynomial time acceptable language then L is also polynomial time decidable language.

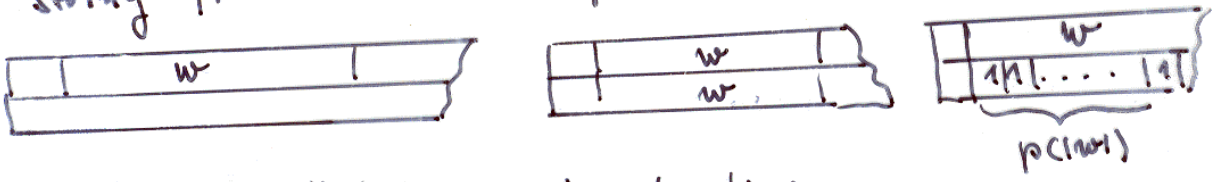
Proof. Let M be T. machine that accepts L without abnormal halting. If it is a danger of abnormal halting, we transform input tape in this way



According assumption there is a polynom $p(n)$, which bonds the number of steps of M accepting $w \in L(M)$ by value $p(|w|)$.

We will construct 2-tape T. machine M' which decides L in polynomial time. M' is a composition of machines M_1 and M_2.

1. M_1 makes copy of input w and its substitution of string $1^{p(|w|)}$ on 2nd tape



it can be shown that M_1 runs in polyn. time.

2. M_2 simulates M with some modifications:

(a) each step of M implies shift on 2nd tape 1 cell right

(b) reaching final states of M implies output ΔYΔΔ

(b) —"— blank on 2nd tape —"— ΔNΔΔ

8.5. Time complexity of nondeterministic machines

Definition 8.3.

We say that a nondeterministic T. machine accepts the language L in polynomial time provided $L = L(M)$ and there is a polynomial $p(x)$ such that for any $w \in L$, M can accept w by a computation involving no more than $p(|w|)$ steps. Futhermore, we define NP to be the class of languages that can be accepted by nondeterministic T. machines in polynomial time.

We can immediately claim that

$$\boxed{P \subseteq NP}$$

The question

$$\boxed{P \overset{?}{=} PN}$$

is one of the most known open problems. It is believed that $P \subset PN$.

Summary:

Polynomial time deterministic computations:

solvable decision problems	=	decidable languages	=	acceptable languages

Polynomial time nondeterministic computations:

solvable decision problems	=	decidable languages	$\overset{?}{=}$	acceptable languages

8.6. Polynomial reduction (also polynomial transformation)

Definition 8.4

A polynomial reduction from a language L_1 over Σ_1, to another language L_2 over Σ_2 is a function f such that

(1) $\forall w \in \Sigma_1^*: \quad w \in L_1 \iff f(w) \in L_2$

(2) f can be computed by T. machine in polynomial time

If there is a polynomial reduction from L_1 to L_2, we say that L_1 reduces to L_2 and write $L_1 \propto L_2$.

Theorem 8.6.

If $L_1 \propto L_2$ and L_2 is in P, then L_1 is in P.

Proof Let M_f be T. machine which computes f and let $p(x)$ is the time complexity of M_f. For any $w \in L_1$, the computation $f(w)$ requires no more than $p(|w|)$ steps and gives the output of the lenght less or equal $p(|w|) + |w|$.

Let M_2 accepts L_2 with complexity $q(x)$. Now consider the composition $\to M_f M_2$. This machine accepts L_1; for any $w \in L_1$, the machine $\to M_f M_2$ makes maximally

$p(|w|) + q(p|w| + |w|)$ steps. Hence $\to M_f M_2$ has polynomial complexity and $L_1 \in P$.

Example 8.

The function $f: \{x,y\}^* \to \{x,y,z\}^*$ defined by $f(v) = vzv$ is a polynomial reduction from

$$L_1 = \{w \mid w \text{ is palidrome in } \{x,y\}^*\}$$

to

$$L_2 = \{wzw^R \mid w \in \{x,y\}^*\}$$

Theorem 8.6. can be use practically to show that given language L is in P (if we know that another language $L' \in P$). Moreover, by „inverse" of Theorem 8.6:

$$\boxed{\text{If } L_1 \propto L_2 \text{ and } L_1 \notin P \text{ then } L_2 \notin P}$$

we can show nonbelongness to P.

8.7. Cook's theorem

Cook's theorem identifies a language (a decision problem) in the class NP to which any other language in NP can be reduced by some polynomial reduction:

SAT problem – the problem of satisfiability

Let $V = \{v_1, v_2, \dots, v_n\}$ be a finite set of Boolean variables (propositions). We suppose truth assignment: $V \to \{true, false\}$

Each v_i or $\overline{v_i}$ (negation of v_i) is called literal.

We define a clause over V to be formula containing literals connected by $\vee$ (or).

Examples of clauses

$$v_1 \vee \overline{v_2}$$

$$v_2 \vee v_3$$

$$\overline{v_1} \vee \overline{v_3} \vee v_2$$

SAT problems is formulated as follows:

> Given a finite set V of variables and a collection of clauses over V, is there a truth assignment that satisfies the clauses?

A given SAT problem can be coded by a single string: Let $V = \{v_1, v_2, \ldots, v_n\}$. Each literal v_i is coded by the string of 0's, but at ith position is p if coded literal is v_i or n if coded literal is $\bar{v}_i$. Each clause is coded as a list of literals separated by /. The SAT-problem is a list of clauses in parenthesis:

Example

The SAT problem over $V = \{v_1, v_2, v_3\}$ with clauses

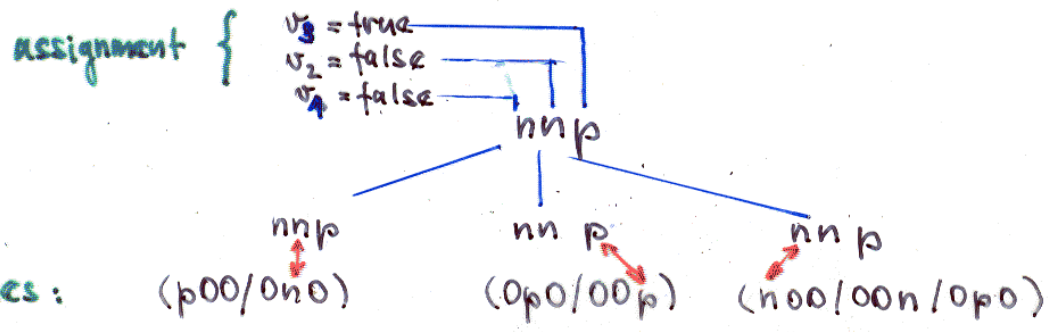
$$v_1 \vee \bar{v}_2 \; ; \; v_2 \vee v_3 \; ; \; \bar{v}_1 \vee \bar{v}_3 \vee v_2$$

is represented by the string

$$\boxed{(p00/0n0)(0p0/00p)(n00/00n/0p0)}$$

Let L_{SAT} be a language of such strings which represent satisfiable sets of clauses. For example, the above string is in L_{SAT} ($v_1 = F$, $v_2 = F$, $v_3 = T$). On the other hand (p00/0p0)(n00/0p0)(p00/0n0)(n00/0n0) is not in L_{SAT}

Let us now represent truth assignments by strings over $\{p, n\}$ where p in ith position indicates that v_i is true and n in j-th position indicates that v_j is false. Then test, if for given truth assignment is satisfiable for given set of clases is simple:



Following this princip of testing (and coding) truth assignments we can construct 2-tape T. machine which is nondeterministic and accepts L_{SAT} in polynomial time. The machine will

1. start computation by checking that its input is a string representing a collection of clauses
2. write, in nondeterministic manner, a string of p's and n's of length equal to the number of variables in the clauses on its second tape
3. move the tape head on the first tape across the input string while testing to see that the truth assignment on its 2nd tape satisfies the clauses being scanned

This process can easily be implemented with polynomial complexity. Hence $L_{SAT} \in NP$.

<u>Theorem 8.7. (Cook's theorem)</u>

If L is ana language in NP, then $L \propto L_{SAT}$.

<u>Proof</u>:

Because $L \in NP$, there is nondeterministic T. machine M and polynomial $p(x)$ such that for any $w \in L$ machine M accepts w in no more than $p(|w|)$ steps. The heart of the proof is a polynomial reduction f from L to L_{SAT}:
for any $w \in L$, the value $f(w)$ will be the set of clauses which are satisfiable if and only if M accepts w

NP-complete languages

Since the discovery of Cook's theorem, many other languages in NP have been found to posses the same pivotal properties as L_{SAT}. (To be polynomial reductions of any other languages in NP) These languages are said to be NP-complete (~~See appendix~~ NPC). If any one of these languages can be accepted by deterministic T. machine in polynomial time, then NP must be equal P. Conversely, if any language in NP can be shown to lie outside of, then all the NP-complete languages must also lie outside of P and search for efficien solution to these problems is useless.

A Sampling of NP Problems

In fact, all the listed problems here are NPC problems.

Formal language and automata theory

1. Two nondeterministic finite automaton over the same alfabet accept different languages.
2. Two regular expressins over the same alfabet represent different languages
3. Given a finete collection of DFA, respond Y if and only if there is a string that is accepted by each of the automaton in the collection.

Number theory

1. Given three positive integers a, b, and c, respond Y if and only if the equation

$$ax^2 + by = c$$

has a solution that consist of positive integers

2. Given three positive integer a, b, and c, respond Y if and only if there is positive integer x that is less than c and $x^2 = a \pmod b$

3. Given a finite collection K of positive integer and another posive integer n, respon Y iff there is a subset of K whose sum is n.

Logic

1. Given a collection of clauses in which each clause contains exactly three literals, respond Y iff there is a true assignment that satisfied the entire collection. This problem is known as 3 SAT.

2. Given an expression consist of variables connected by symbols $\neg, \vee, \wedge, \rightarrow$, respond Y iff there is a <u>model</u> of the expression. (An assignment that evaluates the expression to true value.

Scheduling problems

1. Given a set T of tasks, the amount of time required to complete each individual task, a positive integer n, and a deadline, respond Y iff there is a way of assigning the tasks to n processors so that entire set of tasks will be completed by the deadline.

Graph theory

Definition

Let $\vec{G} = (E, V, \sigma)$ be an oriented graph.

A feedback vertex set is a subset S, $S \subseteq V$ such that any cycle of $\vec{G}$ consist of some vertex from S.

A feedback edge set is a subset F, $F \subseteq E$ such that any cycle of $\vec{G}$ consist of some edge from F

Following are the NPC problems :

1. Is there a clique with k vetexes (k-clique) of given (nororiented) graph G ?
2. (Vertex Cover) Is there a dominating set of k vertexes ?
3. (Hamilton circuit) Is there a Hamilton circuit in graph G ?
4. (Colorability). Is a graph G k-chromatic ?

5. (Feedback vertex set) Is there a feedback vertex set with k vertexes in $\vec{G}$?

6. (Feedback edge set) Is there a feedback edge set with k edges in graph $\vec{G}$?

7. (Directed Hamilton circuit) Is there a Hamilton cycle in oriented graph $\vec{G}$?

Set theory

1. (Set cover)

Let $S_1, S_2, \ldots, S_n$ is a collection of sets. Is there a subset with k sets $S_{i_1}, S_{i_2}, \ldots S_{i_k}$ of the collection such that

$$\bigcup_{j=1}^{k} S_{i_k} = \bigcup_{j=1}^{n} S_j$$

2. (Exact cover)

Let $S_1, S_2, \ldots, S_n$ is a collection of sets. Is there set cover which consist of subset of pair-wise disjoint sets of the collection?

Another approaches to the complexity analysis

Rates (Orders) of Growth

Definition

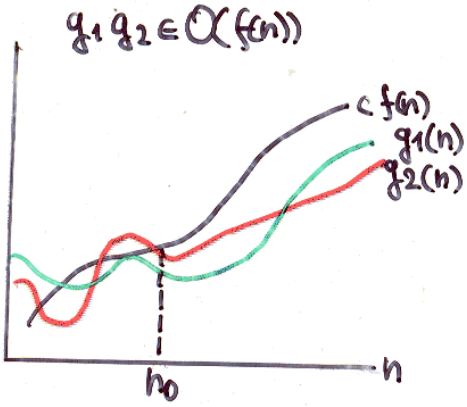
Let $\mathcal{F}$ be a set of functions from $\mathbb{N}$ to $\mathbb{N}$. For given function f from $\mathcal{F}$ we define set of functions $O(f)$, $\Theta(f)$, $\Omega(f)$ called asymtotically upper bound, asymtotically tight bound and asymtotically lower bound respectively in the following way:

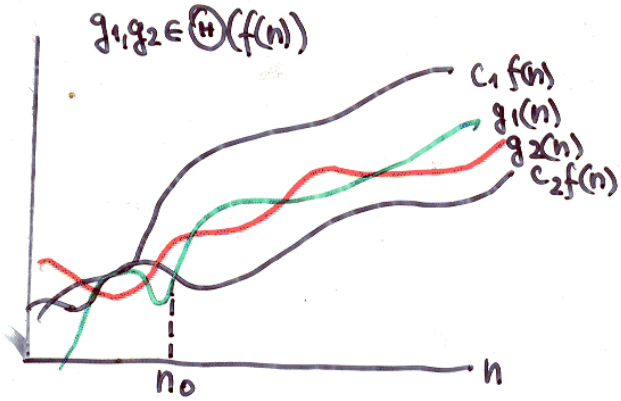
$$O(f(n)) = \{g(n) \mid \text{there are positive constants } c \text{ and } n_0 \text{ such that } 0 \leq g(n) \leq c \cdot f(n) \text{ for all } n \geq n_0\}$$

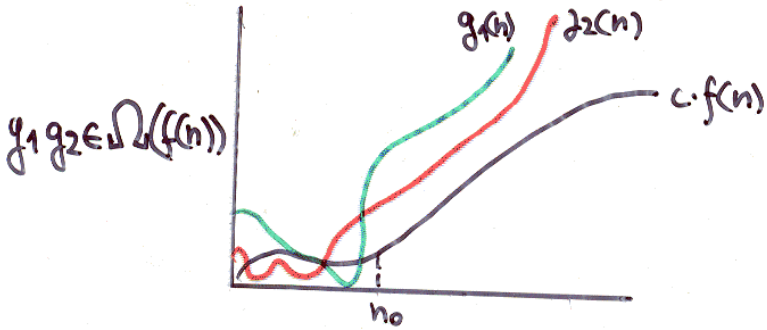
$$\Theta(f(n)) = \{g(n) \mid g(n) \in \mathcal{F} \text{ and there are positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \leq c_1 f(n) \leq g(n) \leq c_2 f(n) \text{ for all } n \geq n_0\}$$

$$\Omega(f(n)) = \{g(n) \mid g(n) \in \mathcal{F} \text{ and there are positive constants } c \text{ and } n_0 \text{ such that } 0 \leq c \cdot f(n) \leq g(n) \text{ for all } n \geq n_0\}$$

The following figures ilustrate the definitions.







<u>Theorem</u>

For any two functions $f(n)$ and $g(n)$ from $\mathcal{F}$ the function $f(n)$ is in $\Theta(g(n))$ iff $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$.

<u>Proof</u>: Follows from definition.

<u>Theorem</u>:

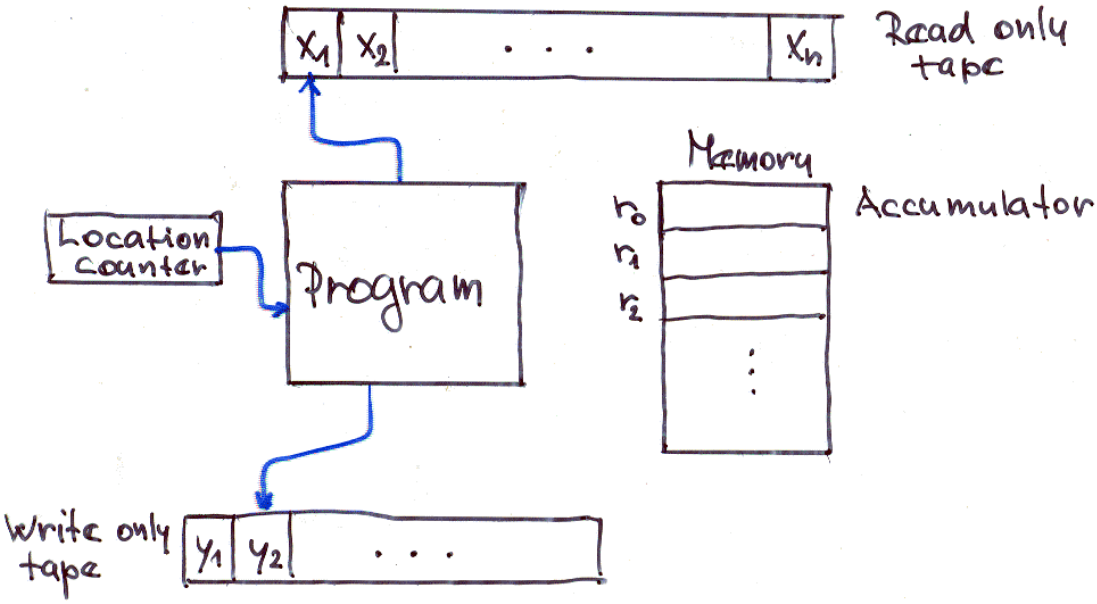
Let $p(n)$ is a polynom with degree d. Then

$$|p(n)| \in \Theta(n^d)$$

Because $\Theta(n^2) \leq \Theta(n^3)$ for example, we will write $\Theta(n^2) \leq \Theta(n^3)$ and we are considering a problem with complexity $\Theta(n^2)$ as less complex than problem with complexity (rate of growth) $\Theta(n^3)$

RAM and RASP models

1. Random access machine – RAM model



Instructions

LOAD	READ
STORE	WRITE
ADD	JUMP
SUB	JGTZ
MULT	JZERO
DIV	HALT

Operands

integers (same as x_i, y_i)

Addressing modes

(a) literal (constant) $=5$

(b) direct address 5

(c) indirect address $*5$

2. Random access stored program machine – RASP model

The same as RAM model but the program is stored in Memory, hence there is no addressing of type (c) – the program can be modified itself.

Theorem: For any RAM program with time complexity $T(n)$ there is a constant k such that time complexity of equivalent RASP program is $k \cdot T(n)$.