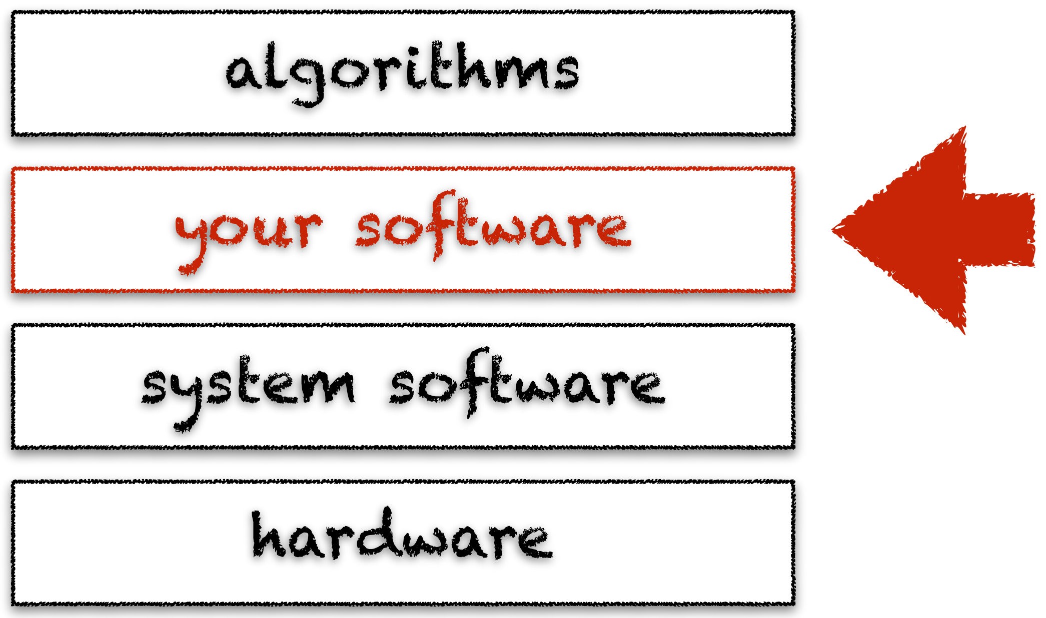
programming basics

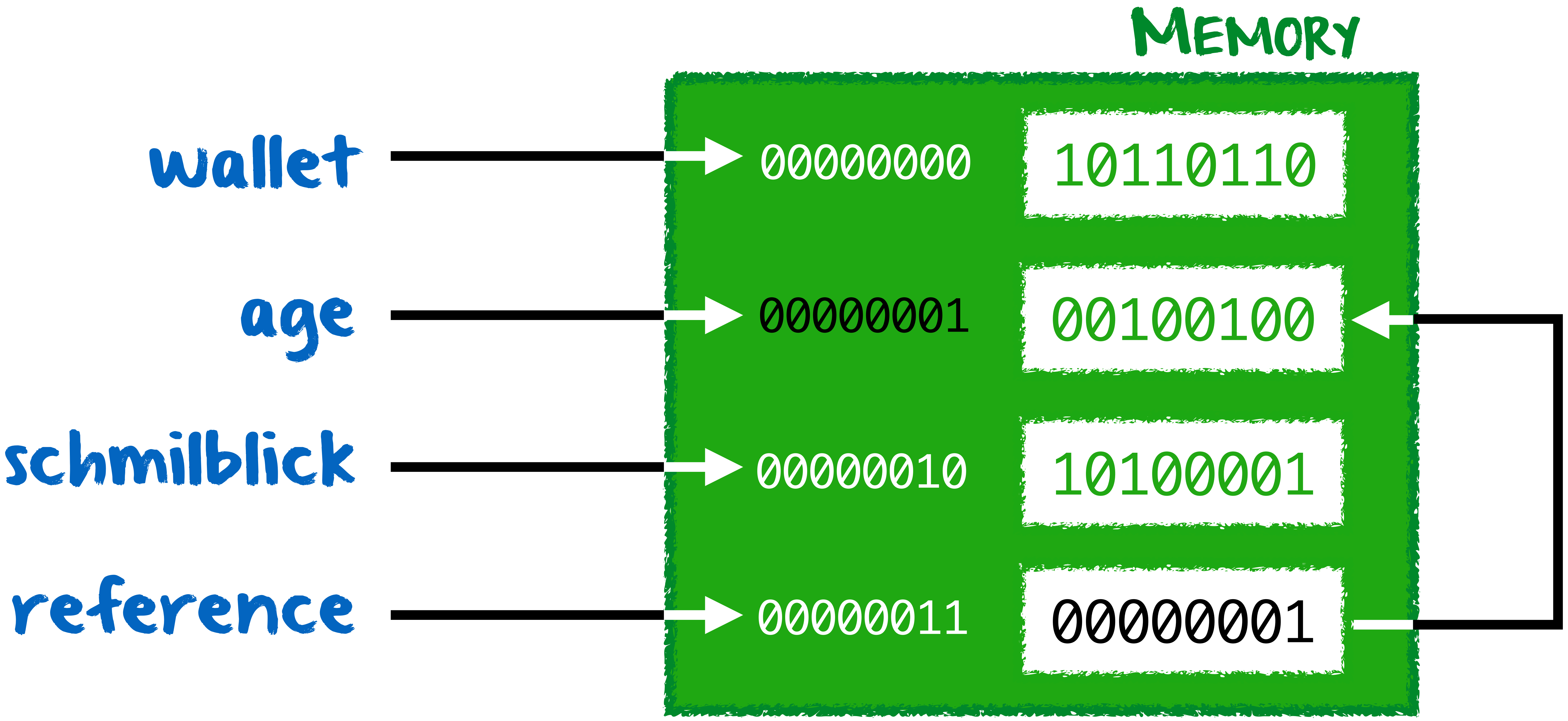
learning objectives



- learn about variables, their types and their values
- learn about different number representations
- learn about functions and how to use them
- learn boolean algebra and conditional branching
- learn about basic text input and output

what's a variable?

in a program, a variable is a symbolic name (also called identifier) associated with a memory location where the value of the variable will be stored

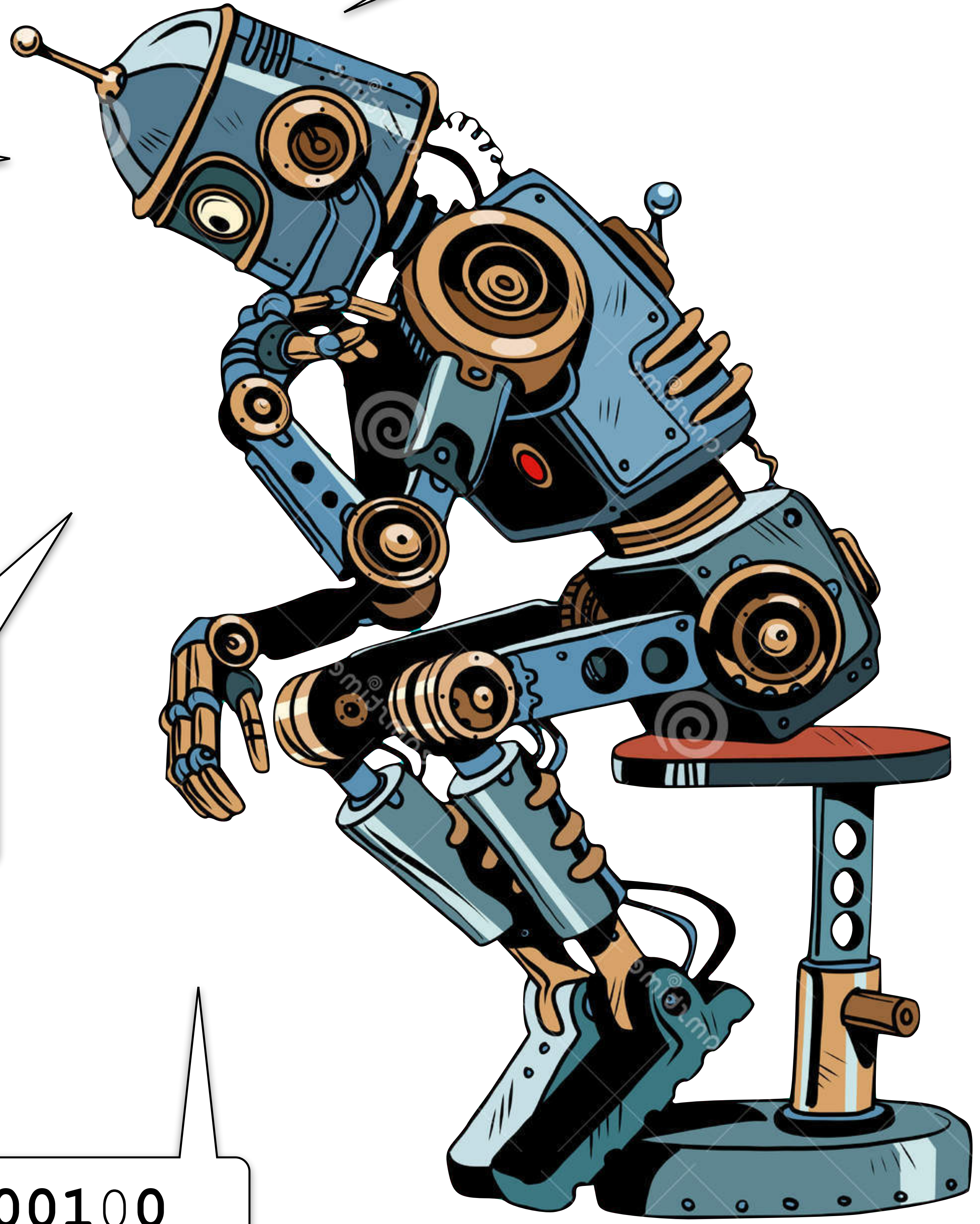


yes but what type of value?

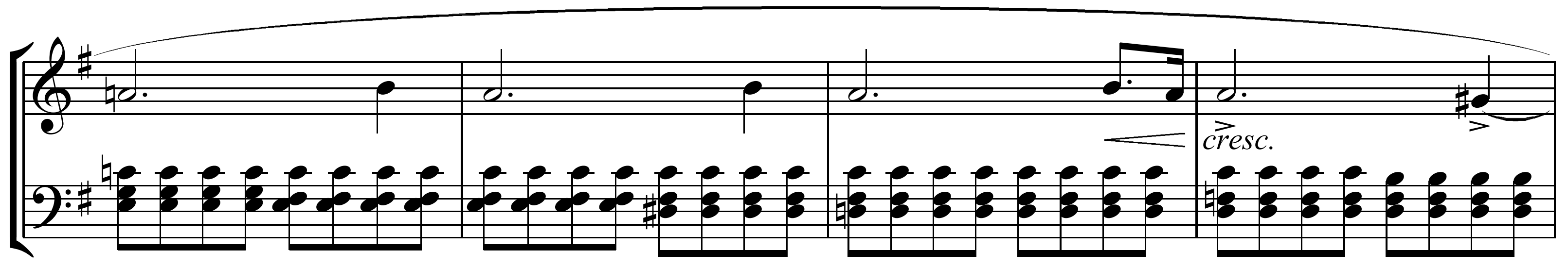
00100101
00101011
00010010
10100100
11001101
00111001
11110011
01010011

$$x^n + y^n = z^n$$

1111001101010011



0010010100101011
1100110100111001
1111001101010011



00100101001010110001001010100100
11001101001110011111001101010011

what's a type?

the **type of a variable** defines **what** will be stored in the memory location, e.g., a boolean, an integer, a character, etc., i.e., **how the bits** in the memory location **will be interpreted**

python	scala	java	swift

`d = 3.14`
`i = 0`
`s = "hello"`
	`var d = 3.14`
	`var i = 0`
	`var s = "hello"`
		`var d = 3.14;`
		`var i = 0;`
		`var s = "Hello";`
			`var d = 3.14`
			`var i = 0`
			`var s = "hello"`

00111110001000000000000000000000 ⇔ 0.15625

1000001 ⇔ 65

1000001 ⇔ ‘A’

0000000 ⇔ false

explicit typing & type inference

as a programmer, you can explicitly define the type of a variable (explicit typing) or let the compiler (or the interpreter) try to infer the type of the variable, typically through initialization (implicit typing)

however, there are cases where type inference is not possible, e.g., in recursive functions

python	scala	java	swift

`i = 0`
`f = 3.14`
`s = "hello"`
	`var i = 0`
	`var d = 3.14`
	`var f = 3.14f`
	`var s = "hello"`
		`var i = 0;`
		`var d = 3.14;`
		`var f = 3.14f;`
		`var s = "Hello";`
			`var i = 0`
			`var d = 3.14`
			`var s = "hello"`

no static typing	`var i : Int = 0`
	`var f : Double = 3.14`
	`var f : Float = 3.14f`
	`var s : String = "hello"`
		`int i = 0;`
		`double d = 3.14;`
		`float f = 3.14f;`
		`String s = "Hello";`
			`var i : Int = 0`
			`var f : Double = 3.14`
			`var f : Float = 3.14`
			`var s : String = "hello"`

static typing vs dynamic typing

the **static type** designates the type of the variable known **at compilation time**

this allows the compiler to **catch** a certain number of **errors** **before** **the execution**

scala

```scala
var i : Int = 0
var d = 3.14
var f = 3.14f
var s = “hello”

f : Float = d
i = d
s = d
```

STOP

the **dynamic type** designates the type of the value contained by a variable **at run time**

this allows the runtime to **catch errors** **during** **the execution**

python

```python
v = 0
v = 3.14
v = “hello”
```

type casting

when you want to assign a value to a variable but the static type and the dynamic type do not match, you can perform an explicit conversion, also known as a type casting

python	scala	java	swift
`d = math.pi`	`var d = math.Pi`	`var d = Math.PI;` 3.141592653589793	`var d : Double.pi`
`i = int(d)`	`var f = d.toFloat`	`var f = (float) d;` 3.1415927	`var i = Int(d)`
`f = float(d)`	`var i = d.toInt`	`var i = (int) d;` 3	`var f = Float(d)`
`s = str(d)`	`var s = d.toString`	`var s = Double.toString(d);` "3.141592653589793"	`var s = String(d)`

number representation

unsigned integers

	bit 7		bit 6		bit 5		bit 4		bit 3		bit 2		bit 1		bit 0
87_{10} =	0×2^7	+	1×2^6	+	0×2^5	+	1×2^4	+	0×2^3	+	1×2^2	+	1×2^1	+	1×2^0
87_{10} =	0×128	+	1×64	+	0×32	+	1×16	+	0×8	+	1×4	+	1×2	+	1×1
87_{10} =	0		1		0		1		0		1		1		1

$87_{10} = 01010111_2$

range = $[0_2, 11111111_2] = [0_{10}, 255_{10}]$

signed integers with signed magnitude

	Bit 7	bit 6		bit 5		bit 4		bit 3		bit 2		bit 1		bit 0
87_{10} =	0	1×2^6	+	0×2^5	+	1×2^4	+	0×2^3	+	1×2^2	+	1×2^1	+	1×2^0
87_{10} =	0	1		0		1		0		1		1		1
-87_{10} =	1	1×64	+	0×32	+	1×16	+	0×8	+	1×4	+	1×2	+	1×1
-87_{10} =	1	1		0		1		0		1		1		1

$87_{10} = 01010111_2$
$-87_{10} = 11010111_2$

Bit 7 is the sign bit

0 ⇔ +
1 ⇔ −

range = $[-127_{10}, +127_{10}]$

two ways to represent zero:
$+0_{10} = 00000000_2$
$-0_{10} = 10000000_2$

number representation

signed integers with one complement															
	Bit 7	bit 6		bit 5		bit 4		bit 3		bit 2		bit 1		bit 0
87_{10} =	0	1×2^6	+	0×2^5	+	1×2^4	+	0×2^3	+	1×2^2	+	1×2^1	+	1×2^0
87_{10} =	0	1		0		1		0		1		1		1
	not ⬇	not ⬇		not ⬇		not ⬇		not ⬇		not ⬇		not ⬇		not ⬇
-87_{10} =	1	0		1		0		1		0		0		0

$87_{10} = \mathbf{0}1010111_2$
$-87_{10} = \mathbf{1}0101000_2$

Bit 7 is the sign bit

0 ⇔ +
1 ⇔ −

range = $[-127_{10}, +127_{10}]$

two ways to represent zero:
$+0_{10} = 00000000_2$
$-0_{10} = 11111111_2$

number representation

signed integers with two complement															
	Bit 7	bit 6		bit 5		bit 4		bit 3		bit 2		bit 1		bit 0
87_{10} =	0	1×2^6	+	0×2^5	+	1×2^4	+	0×2^3	+	1×2^2	+	1×2^1	+	1×2^0
87_{10} =	0	1		0		1		0		1		1		1
	not ⬇	not ⬇		not ⬇		not ⬇		not ⬇		not ⬇		not ⬇		not ⬇
	1	0		1		0		1		0		0		0
														+1 ⬇
-87_{10} =	1	0		1		0		1		0		0		1
	-1×2^7	0×2^6	+	1×2^5	+	0×2^4	+	1×2^3	+	0×2^2	+	0×2^1	+	1×2^0
-87_{10} =	-1×128		+	1×32			+	1×8					+	1×1

$87_{10} = \mathbf{0}1010111_2$
$-87_{10} = \mathbf{1}0101001_2$

Bit 7 is the sign bit

0 ⇔ +
1 ⇔ −

range = $[-128_{10}, +127_{10}]$

only one way to represent zero:
$0_{10} = 00000000_2$

number representation

signed integers with two complement – further examples

	Bit 7 sign	bit 6 64	bit 5 32	bit 4 16	bit 3 8	bit 2 4	bit 1 2	bit 0 1
44_{10} =	0	0	1	0	1	1	0	0
	not ⬇	not ⬇	not ⬇	not ⬇	not ⬇	not ⬇	not ⬇	not ⬇
	1	1	0	1	0	0	1	1
								+1 ⬇
-44_{10} =	1	1	0	1	0	1	0	0

	Bit 7 sign	bit 6 64	bit 5 32	bit 4 16	bit 3 8	bit 2 4	bit 1 2	bit 0 1
0_{10} =	0	0	0	0	0	0	0	0
	not ⬇	not ⬇	not ⬇	not ⬇	not ⬇	not ⬇	not ⬇	not ⬇
	1	1	1	1	1	1	1	1
								+1 ⬇
-0_{10} =	0	0	0	0	0	0	0	0

number representation

only a small subset of the infinite set of real numbers can be represented in a computer, which has a finite memory space

floating point principle

$$\text{sign} \times \text{mantissa} \times \text{base}^{\text{exponent}}$$

$$-3.14159 = -1 \times 314159 \times 10^{-5}$$

in a computer, the base is 2

number representation

floating point single precision

sign **exponent** (8 bits) **mantissa** (23 bits)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

$$\text{value} = (-1)^{\text{sign}} \times \left(1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}\right) \times 2^{(e-127)}$$

$$\text{sign} = b_{31} = 0 \implies (-1)^{\text{sign}} = (-1)^0 = +1 \in \{-1, +1\}$$

$$e = b_{30} b_{29} \dots b_{23} = \sum_{i=0}^{7} b_{23+i} 2^{+i} = 124 \in \{1, \dots, (2^8 - 1) - 1\} = \{1, \dots, 254\}$$

$$2^{(e-127)} = 2^{124-127} = 2^{-3} \in \{2^{-126}, \dots, 2^{127}\}$$

$$1.b_{22} b_{21} \dots b_0 = 1 + \sum_{i=1}^{23} b_{23-i} 2^{-i} = 1 + 1 \cdot 2^{-2} = 1.25 \in \{1, 1 + 2^{-23}, \dots, 2 - 2^{-23}\} \subset [1; 2 - 2^{-23}] \subset [1; 2)$$

$$\text{value} = (+1) \times 1.25 \times 2^{-3} = +0.15625$$

character representation

ASCII

	_0	_1	_2	_3	_4	_5	_6	_7	_8	_9	_A	_B	_C	_D	_E	_F
0_	NUL 0000 *0*	SOH 0001 *1*	STX 0002 *2*	ETX 0003 *3*	EOT 0004 *4*	ENQ 0005 *5*	ACK 0006 *6*	BEL 0007 *77*	BS 0008 *8*	HT 0009 *9*	LF 000A *10*	VT 000B *11*	FF 000C *12*	CR 000D *13*	SO 000E *14*	SI 000F *15*
1_	DLE 0010 *16*	DC1 0011 *17*	DC2 0012 *18*	DC3 0013 *19*	DC4 0014 *20*	NAK 0015 *21*	SYN 0016 *22*	ETB 0017 *23*	CAN 0018 *24*	EM 0019 *25*	SUB 001A *26*	ESC 001B *27*	FS 001C *28*	GS 001D *29*	RS 001E *30*	US 001F *31*
2_	SP 0020 *32*	! 0021 *33*	" 0022 *34*	# 0023 *35*	$ 0024 *36*	% 0025 *37*	& 0026 *38*	' 0027 *39*	(0028 *40*	) 0029 *41*	* 002A *42*	+ 002B *43*	, 002C *44*	- 002D *45*	. 002E *46*	/ 002F *47*
3_	0 0030 *48*	1 0031 *49*	2 0032 *50*	3 0033 *51*	4 0034 *52*	5 0035 *53*	6 0036 *54*	7 0037 *55*	8 0038 *56*	9 0039 *57*	: 003A *58*	; 003B *59*	< 003C *60*	= 003D *61*	> 003E *62*	? 003F *63*
4_	@ 0040 *64*	A 0041 *65*	B 0042 *66*	C 0043 *67*	D 0044 *68*	E 0045 *69*	F 0046 *70*	G 0047 *71*	H 0048 *72*	I 0049 *73*	J 004A *74*	K 004B *75*	L 004C *76*	M 004D *77*	N 004E *78*	O 004F *79*
5_	P 0050 *80*	Q 0051 *81*	R 0052 *82*	S 0053 *83*	T 0054 *84*	U 0055 *85*	V 0056 *86*	W 0057 *87*	X 0058 *88*	Y 0059 *89*	Z 005A *90*	[005B *91*	\ 005C *92*	] 005D *93*	^ 005E *94*	_ 005F *95*
6_	` 0060 *96*	a 0061 *97*	b 0062 *98*	c 0063 *99*	d 0064 *100*	e 0065 *101*	f 0066 *102*	g 0067 *103*	h 0068 *104*	i 0069 *105*	j 006A *106*	k 006B *107*	l 006C *108*	m 006D *109*	n 006E *110*	o 006F *111*
7_	p 0070 *112*	q 0071 *113*	r 0072 *114*	s 0073 *115*	t 0074 *116*	u 0075 *117*	v 0076 *118*	w 0077 *119*	x 0078 *120*	y 0079 *121*	z 007A *122*	{ 007B *123*	| 007C *124*	} 007D *125*	~ 007E *126*	DEL 007F *127*

Letter · Number · Punctuation · Symbol · Other · undefined · Changed from 1963 version

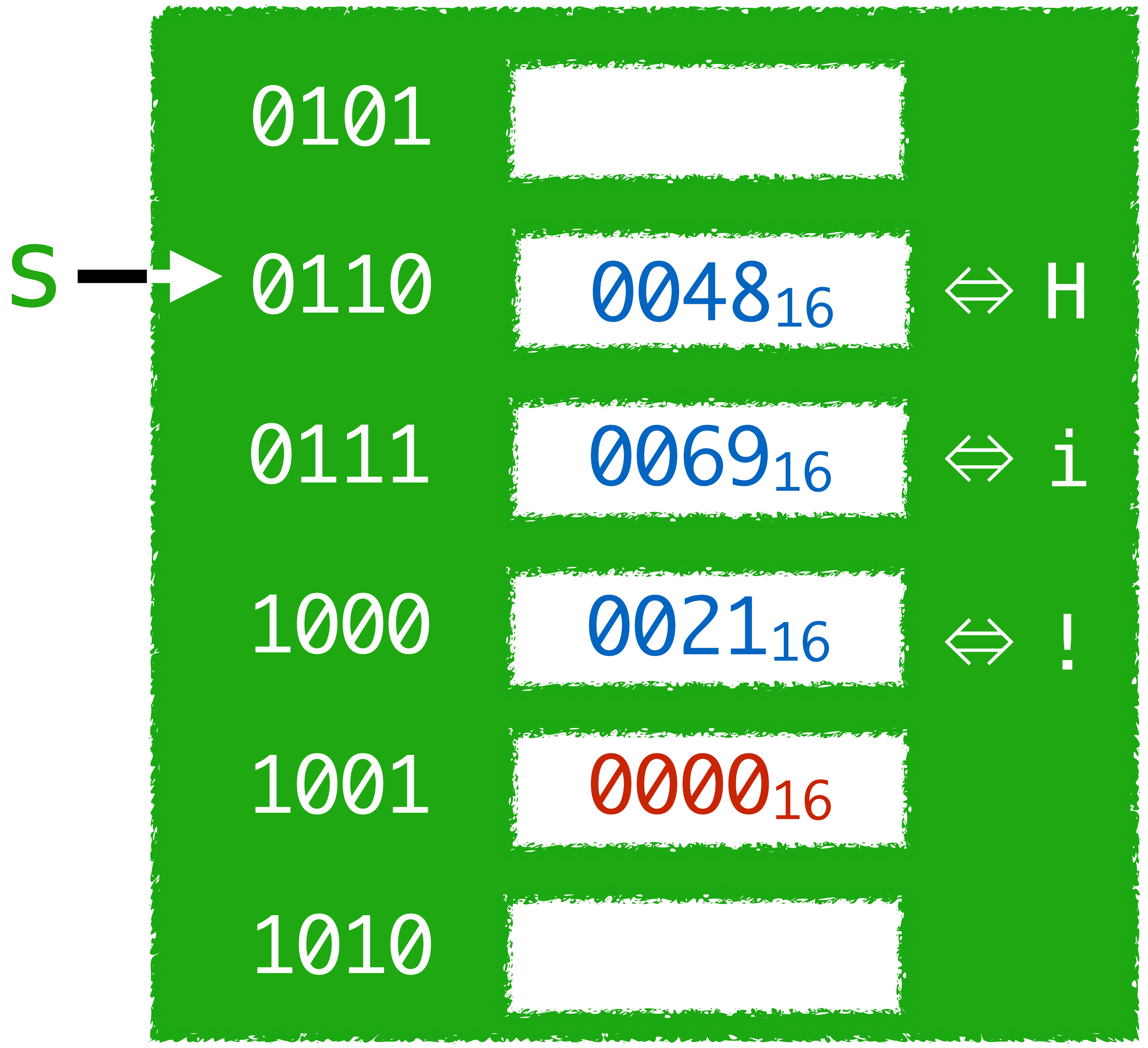
UTF-8

	_0	_1	_2	_3	_4	_5	_6	_7	_8	_9	_A	_B	_C	_D	_E	_F
0_	NUL 0000	SOH 0001	STX 0002	ETX 0003	EOT 0004	ENQ 0005	ACK 0006	BEL 0007	BS 0008	HT 0009	LF 000A	VT 000B	FF 000C	CR 000D	SO 000E	SI 000F
1_	DLE 0010	DC1 0011	DC2 0012	DC3 0013	DC4 0014	NAK 0015	SYN 0016	ETB 0017	CAN 0018	EM 0019	SUB 001A	ESC 001B	FS 001C	GS 001D	RS 001E	US 001F
2_	SP 0020	! 0021	" 0022	# 0023	$ 0024	% 0025	& 0026	' 0027	(0028	) 0029	* 002A	+ 002B	, 002C	- 002D	. 002E	/ 002F
3_	0 0030	1 0031	2 0032	3 0033	4 0034	5 0035	6 0036	7 0037	8 0038	9 0039	: 003A	; 003B	< 003C	= 003D	> 003E	? 003F
4_	@ 0040	A 0041	B 0042	C 0043	D 0044	E 0045	F 0046	G 0047	H 0048	I 0049	J 004A	K 004B	L 004C	M 004D	N 004E	O 004F
5_	P 0050	Q 0051	R 0052	S 0053	T 0054	U 0055	V 0056	W 0057	X 0058	Y 0059	Z 005A	[005B	\ 005C	] 005D	^ 005E	_ 005F
6_	` 0060	a 0061	b 0062	c 0063	d 0064	e 0065	f 0066	g 0067	h 0068	i 0069	j 006A	k 006B	l 006C	m 006D	n 006E	o 006F
7_	p 0070	q 0071	r 0072	s 0073	t 0074	u 0075	v 0076	w 0077	x 0078	y 0079	z 007A	{ 007B	| 007C	} 007D	~ 007E	DEL 007F
8_	• +00	• +01	• +02	• +03	• +04	• +05	• +06	• +07	• +08	• +09	• +0A	• +0B	• +0C	• +0D	• +0E	• +0F
9_	• +10	• +11	• +12	• +13	• +14	• +15	• +16	• +17	• +18	• +19	• +1A	• +1B	• +1C	• +1D	• +1E	• +1F
A_	• +20	• +21	• +22	• +23	• +24	• +25	• +26	• +27	• +28	• +29	• +2A	• +2B	• +2C	• +2D	• +2E	• +2F
B_	• +30	• +31	• +32	• +33	• +34	• +35	• +36	• +37	• +38	• +39	• +3A	• +3B	• +3C	• +3D	• +3E	• +3F
2 C_	2 0000	2 0040	Latin 0080	Latin 00C0	Latin 0100	Latin 0140	Latin 0180	Latin 01C0	Latin 0200	IPA 0240	IPA 0280	IPA 02C0	accents 0300	accents 0340	Greek 0380	Greek 03C0
2 D_	Cyril 0400	Cyril 0440	Cyril 0480	Cyril 04C0	Cyril 0500	Armeni 0540	Hebrew 0580	Hebrew 05C0	Arabic 0600	Arabic 0640	Arabic 0680	Arabic 06C0	Syriac 0700	Arabic 0740	Thaana 0780	N'Ko 07C0
3 E_	Indic 0800	Misc. 1000	Symbol 2000	Kana… 3000	CJK 4000	CJK 5000	CJK 6000	CJK 7000	CJK 8000	CJK 9000	Asian A000	Hangul B000	Hangul C000	Hangul D000	PUA E000	Forms F000
4 F_	SMP… 10000	40000	80000	SSP… C0000	SPU… 100000	4 140000	4 180000	4 1C0000	5 200000	5 1000000	5 2000000	5 3000000	6 4000000	6 40000000		

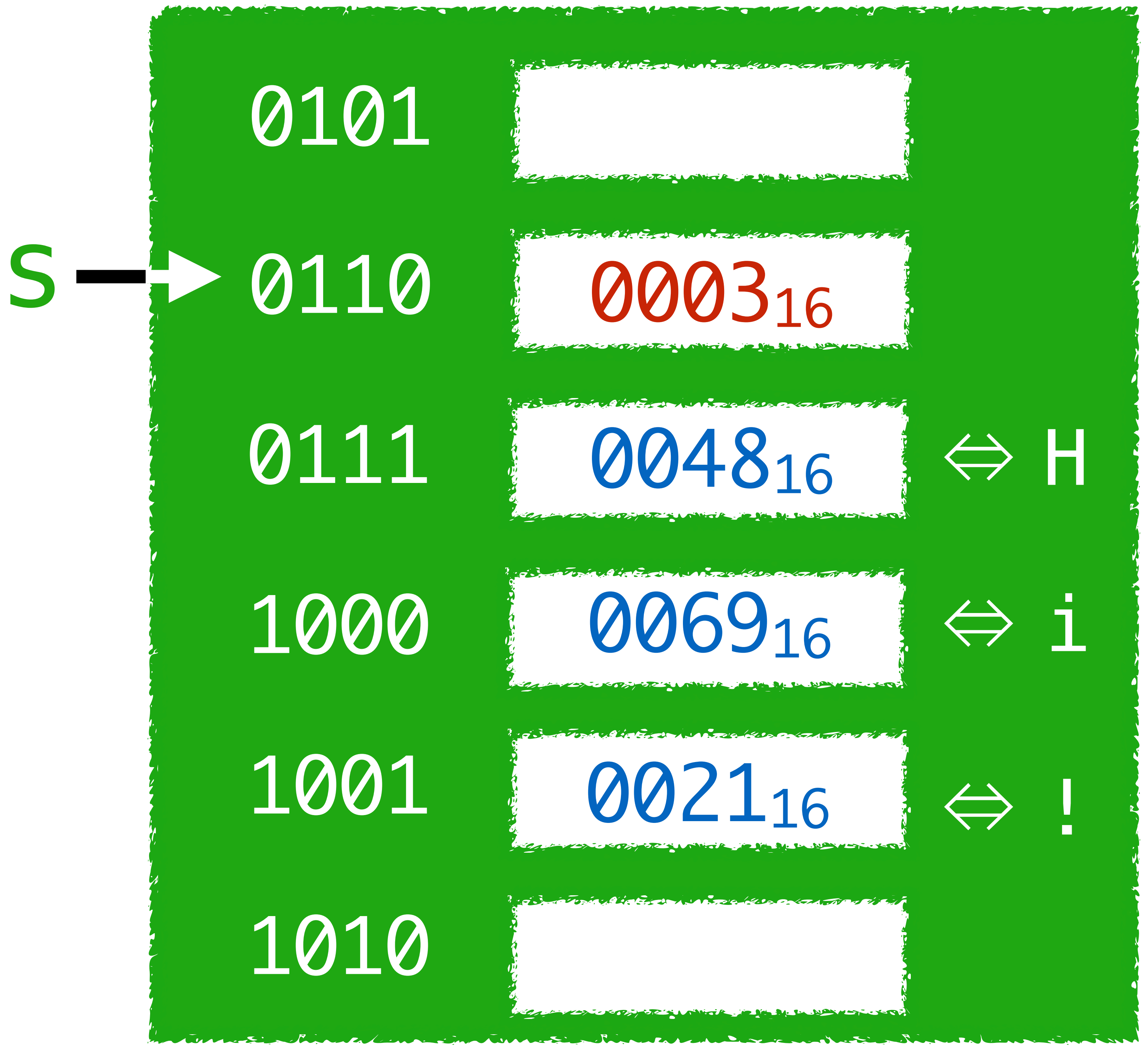
string representation

```
var s = "Hi!"
```

null-terminated string



length-prefixed string



what's a constant?

a constant is simply a
variable that cannot... vary 😜

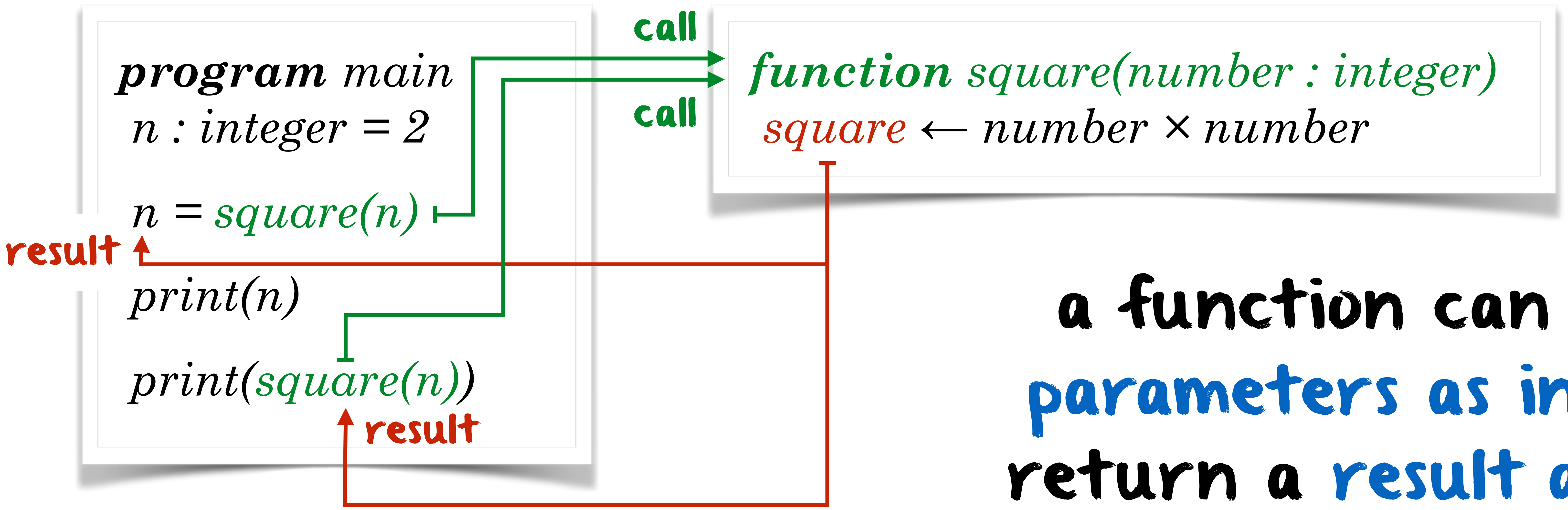
python	scala	java	swift

no constant	`val d : Double = math.Pi`
	`val i = 0`
	`val s = "hello"`
	
	`d = 1.0`
	`i = 1`
	`s = "bye"`
	STOP
		`final var d = Math.PI;`
		`final var i = 0;`
		`final var d = "hello";`
		
		`d = 1.0;`
		`i = 1;`
		`s = "bye";`
		STOP
			`let d : Double.pi`
			`let i = 0`
			`let s = "hello"`
			
			`d = 1.0`
			`i = 1`
			`s = "bye"`
			STOP

what's a function?

in a program, a function is a symbolic name (identifier) associated with a sequence of instructions that performs a specific task

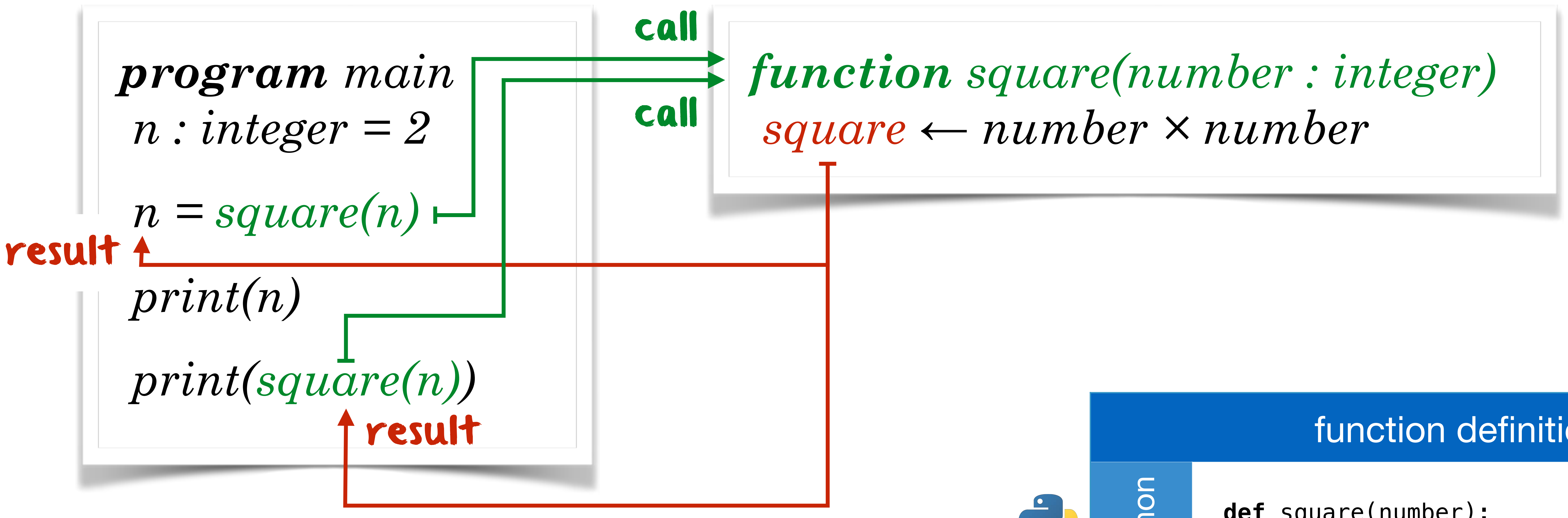
once defined, a function can then be called in programs wherever that particular task should be performed



a function can receive parameters as input and return a result as output

function ⇔ procedure ⇔ routine ⇔ subroutine ⇔ subprogram ⇔ method

what's a function?



	function definition	function call
python	`def square(number):` `  return number * number`	`result = square(2)`

scala	`def square(number : Int) : Int = {`
	`  number * number`
	`}`
		`var result = square (2)`

java	`public int square(int number) {`
	`  return number * number`
	`}`
		`int result = square(2)`

swift	`func square(number:Int) -> Int {`
	`  return number * number`
	`}`
		`var result = square (2)`

logic

the intellectual tool
for reasoning about
the truth and falsity
of statements

logic & programming

most programming languages, support **boolean variables**, which can take values $\in$ {`true`,`false`}

in some low-level languages, integer numbers are used for the same purpose, e.g,. with:

```
p = false    ⇔  p = 0
q = true     ⇔  q = 1 (sometimes  q = true   ⇔  q ≠ 0 )
```

when combined with operators $\wedge$, $\vee$ and $\neg$, boolean variables constitute an algebra used in **conditional branching**

where:

$\neg \Leftrightarrow$ not

$\vee \Leftrightarrow$ or

$\wedge \Leftrightarrow$ and

boolean algebra

assume that p , q and r are boolean variables (or statements) and that $T = true$, $F = false$, we have:

p	$\neg p$
F	T
T	F

p	q	$p \wedge q$
F	F	F
F	T	F
T	F	F
T	T	T

p	q	$p \vee q$
F	F	F
F	T	T
T	F	T
T	T	T

$\neg \Leftrightarrow$ **not**
$\vee \Leftrightarrow$ **or**
$\wedge \Leftrightarrow$ **and**

python	scala	java	swift
`a = False`	`var a = false`	`var a = false;`	`var a = false`
`b = True`	`var b = true`	`var b = true;`	`var b = true`
`c = a and b`	`var c = a && b`	`var c = a && b;`	`var c = a && b`
`c = a or b`	`c = a || b`	`c = a || b;`	`c = a || b`
`c = not a`	`c = !a`	`c = !a;`	`c = !a`

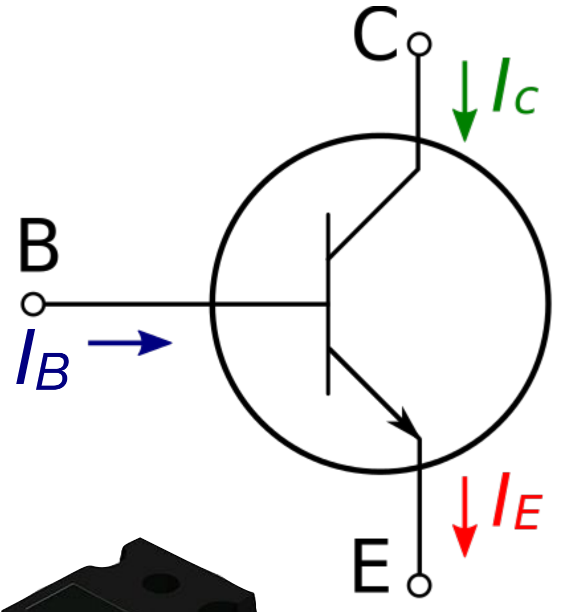
some rules

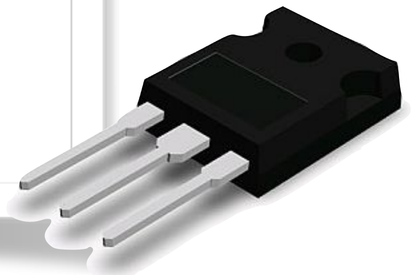
Associative Rules:	$(p \wedge q) \wedge r \Leftrightarrow p \wedge (q \wedge r)$	$(p \vee q) \vee r \Leftrightarrow p \vee (q \vee r)$
Distributive Rules:	$p \wedge (q \vee r) \Leftrightarrow (p \wedge q) \vee (p \wedge r)$	$p \vee (q \wedge r) \Leftrightarrow (p \vee q) \wedge (p \vee r)$
Idempotent Rules:	$p \wedge p \Leftrightarrow p$	$p \vee p \Leftrightarrow p$
Double Negation:	$\neg\neg p \Leftrightarrow p$	
DeMorgan's Rules:	$\neg(p \wedge q) \Leftrightarrow \neg p \vee \neg q$	$\neg(p \vee q) \Leftrightarrow \neg p \wedge \neg q$
Commutative Rules:	$p \wedge q \Leftrightarrow q \wedge p$	$p \vee q \Leftrightarrow q \vee p$
Absorption Rules:	$p \vee (p \wedge q) \Leftrightarrow p$	$p \wedge (p \vee q) \Leftrightarrow p$
Bound Rules:	$p \wedge F \Leftrightarrow F \quad p \wedge T \Leftrightarrow p$	$p \vee T \Leftrightarrow T \quad p \vee F \Leftrightarrow p$
Negation Rules:	$p \wedge (\neg p) \Leftrightarrow F$	$p \vee (\neg p) \Leftrightarrow T$

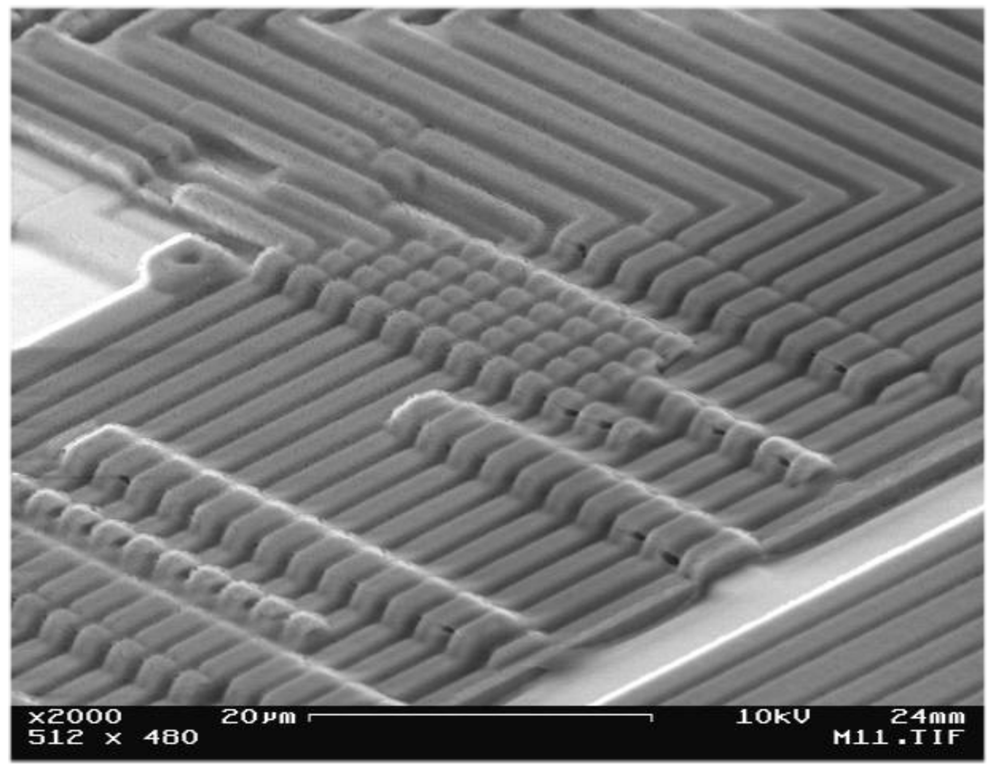
transistors & boolean algebra

the example of the "and" and "or" gates

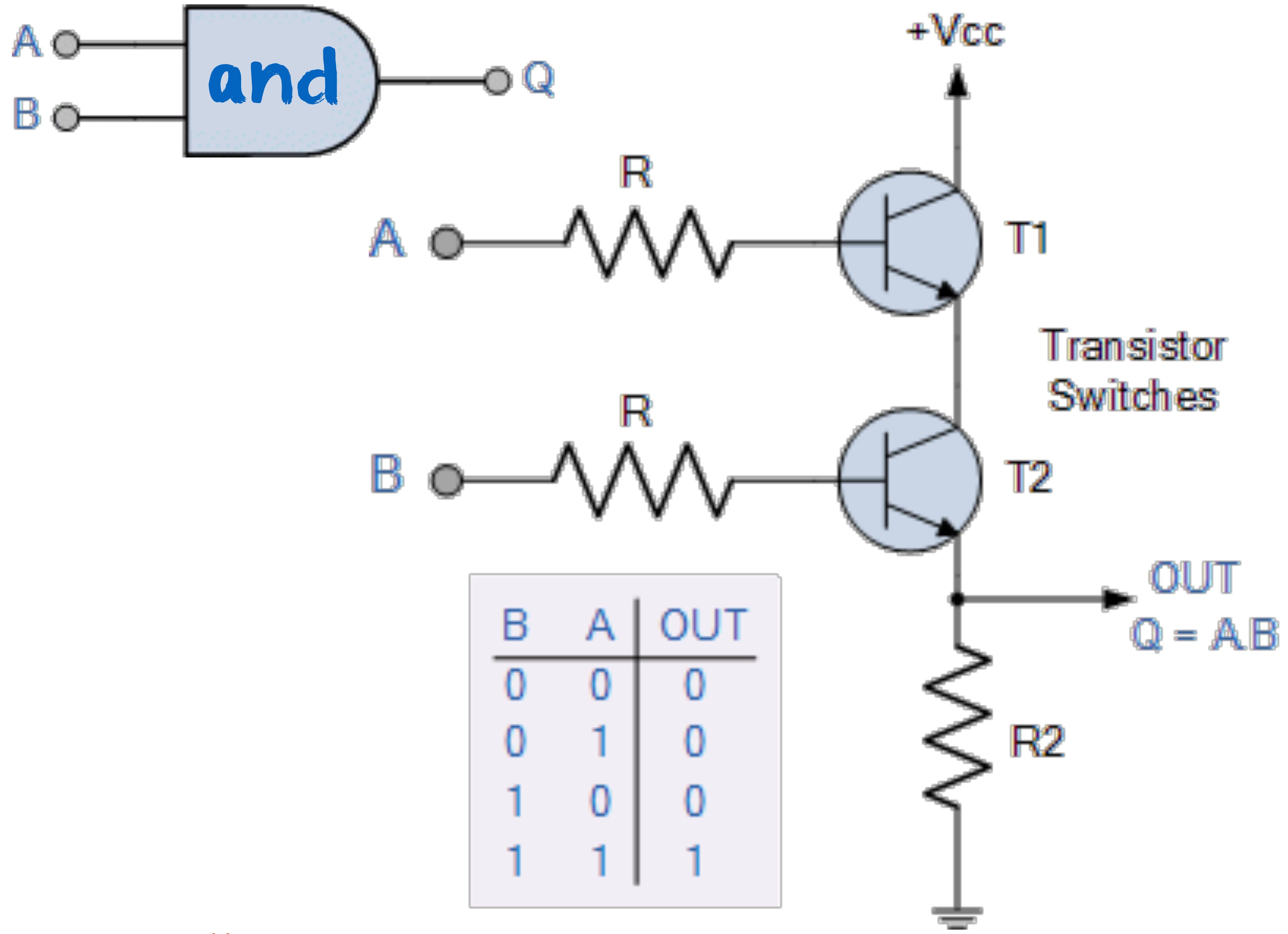
a **transistor** is a **device** that can **amplify or switch** an electrical **current**, using three layers of a **semiconductor material**



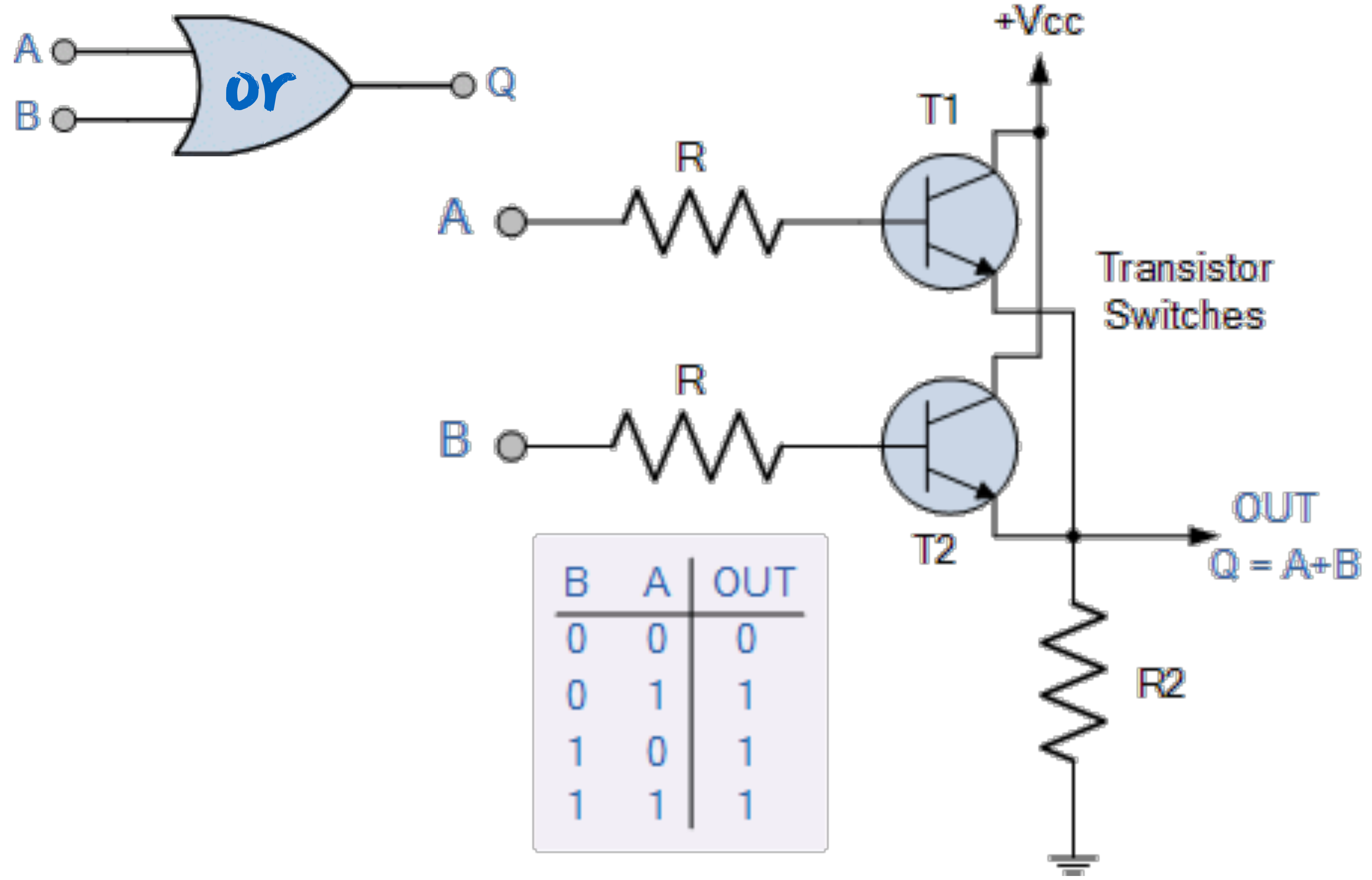




10 μm	1971
6 μm	1974
3 μm	1977
1.5 μm	1981
1 μm	1984
800 nm	1987
600 nm	1990
350 nm	1993
250 nm	1996
180 nm	1999
130 nm	2001
90 nm	2003
65 nm	2005
45 nm	2007
32 nm	2009
22 nm	2012
14 nm	2014
10 nm	2016
7 nm	2018
5 nm	2019
3 nm	2021



B	A	OUT
0	0	0
0	1	0
1	0	0
1	1	1



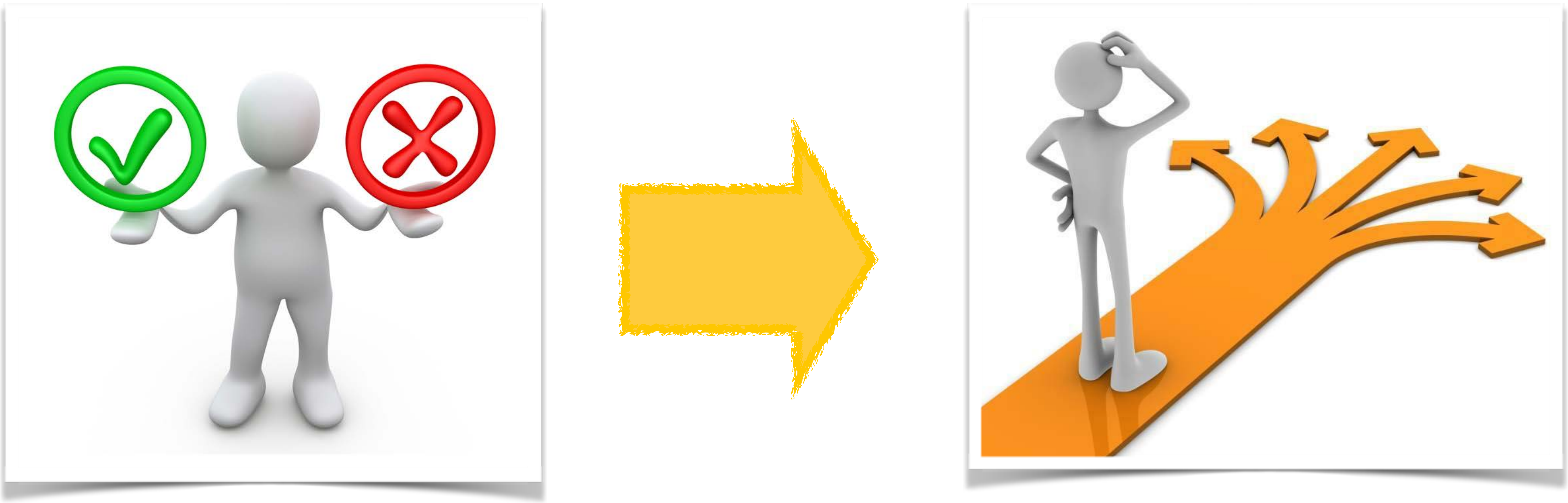
B	A	OUT
0	0	0
0	1	1
1	0	1
1	1	1

source: https://www.electronics-tutorials.ws

from boolean algebra to conditional branching

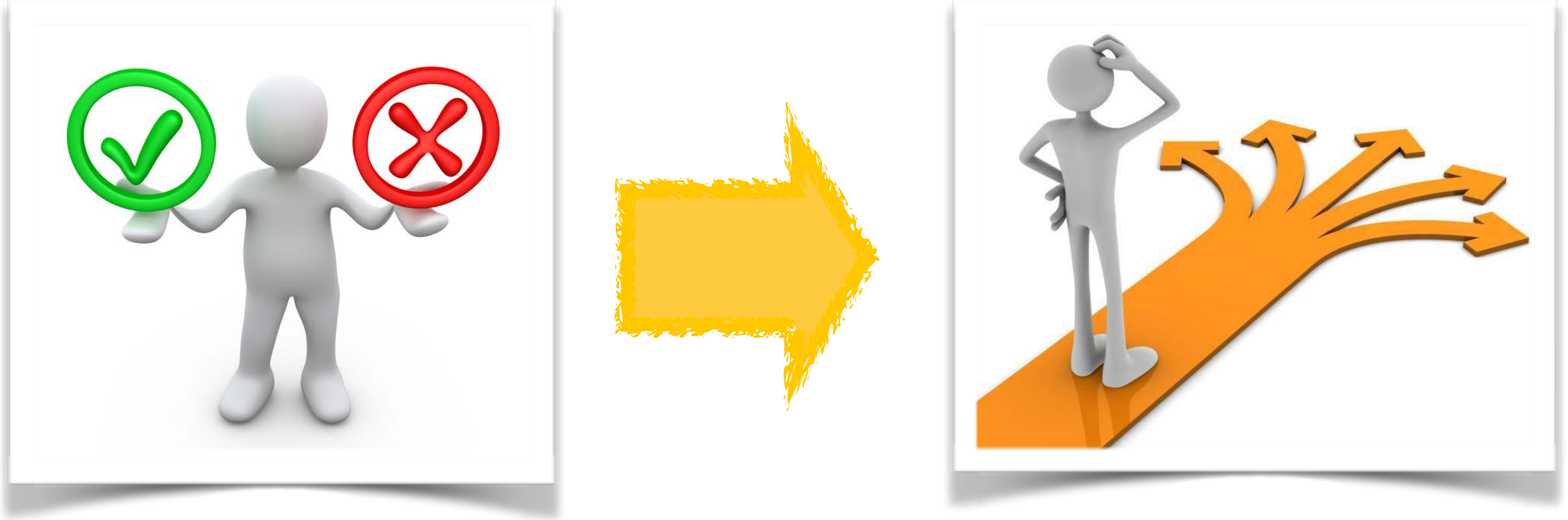
example

write a function that checks whether a given year (passed as parameter) is a leap year or not



Leap years are multiples of 4, and they can only be multiples of 100 if they are also multiples of 400

```
function isLeap(year : integer)
if  year mod 400 = 0
    isLeap ← true
else if  year mod 100 = 0
    isLeap ← false
else if  year mod 4 = 0
    isLeap ← true
else  isLeap ← false
```



conditional branching

```
function isLeap(year : integer)
if  ((year mod 4 = 0) ∧ (year mod 100 ≠ 0))  ∨  (year mod 400)
    isLeap ← true
else
    isLeap ← false
```

```
function isLeap(year : integer)
isLeap ← ((year mod 4 = 0) ∧ (year mod 100 ≠ 0))  ∨  (year mod 400)
```

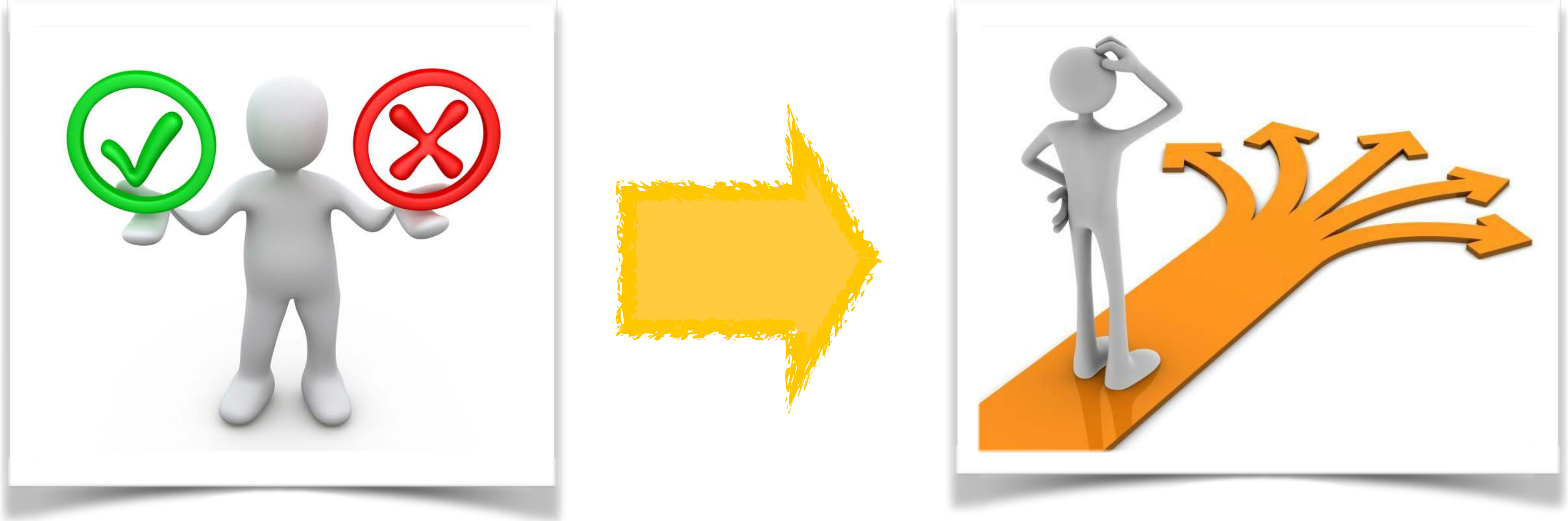
conditional branching

python

```
def isLeap(year):
    if year % 400 == 0 : return True
    elif year % 100 == 0 : return False
    elif year % 4 == 0 : return True
    return False
```

```
def isLeap(year):
    if (year % 4 == 0) and (year % 100 != 0) or (year % 400 == 0) : return True
    return False
```

```
def isLeap(year):
    return (year % 4 == 0) and (year % 100 != 0) or (year % 400 == 0)
```



conditional branching

scala

```scala
def isLeap(year : Int) : Boolean = {
  if   (year % 400 == 0) true
  else if (year % 100 == 0) false
  else if (year % 4 == 0) true
  else false
}
```

```scala
def isLeap(year : Int) : Boolean = {
  if  ((year % 4 == 0) && (year % 100 != 0) || (year % 400 == 0)) true
  else false
}
```

```scala
def isLeap(year : Int) : Boolean =
(year % 4 == 0) && (year % 100 != 0) || (year % 400 == 0)
```

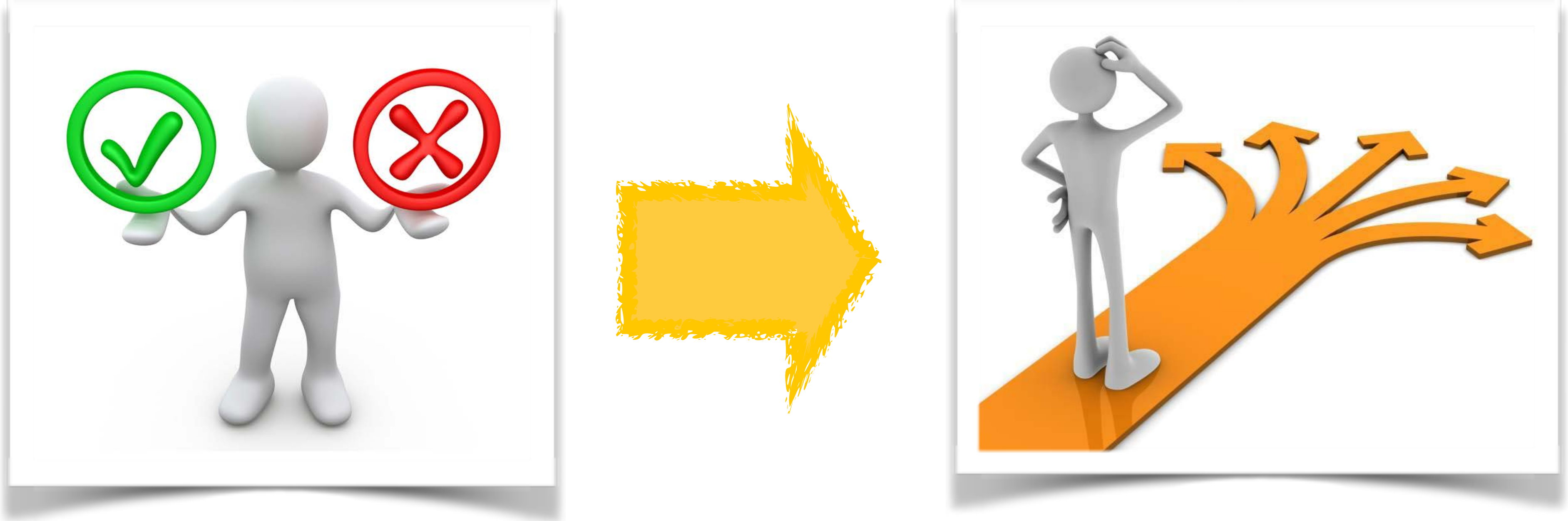
conditional branching

java

```
public class LeapYear {
  public static boolean isLeap(int year) {
    if (year % 400 == 0) return true;
    if (year % 100 == 0) return false;
    if (year % 4 == 0) return true;
    return false;
  }
}
```

```
public class LeapYear {
  public static boolean isLeap(int year) {
    if ((year % 4 == 0) && (year % 100 != 0) || (year % 400 == 0))
      return true;
    else return false;
}
```

```
public class LeapYear {
  public static boolean isLeap(int year) {
    return (year % 4 == 0) && (year % 100 != 0) || (year % 400 == 0);
  }
}
```



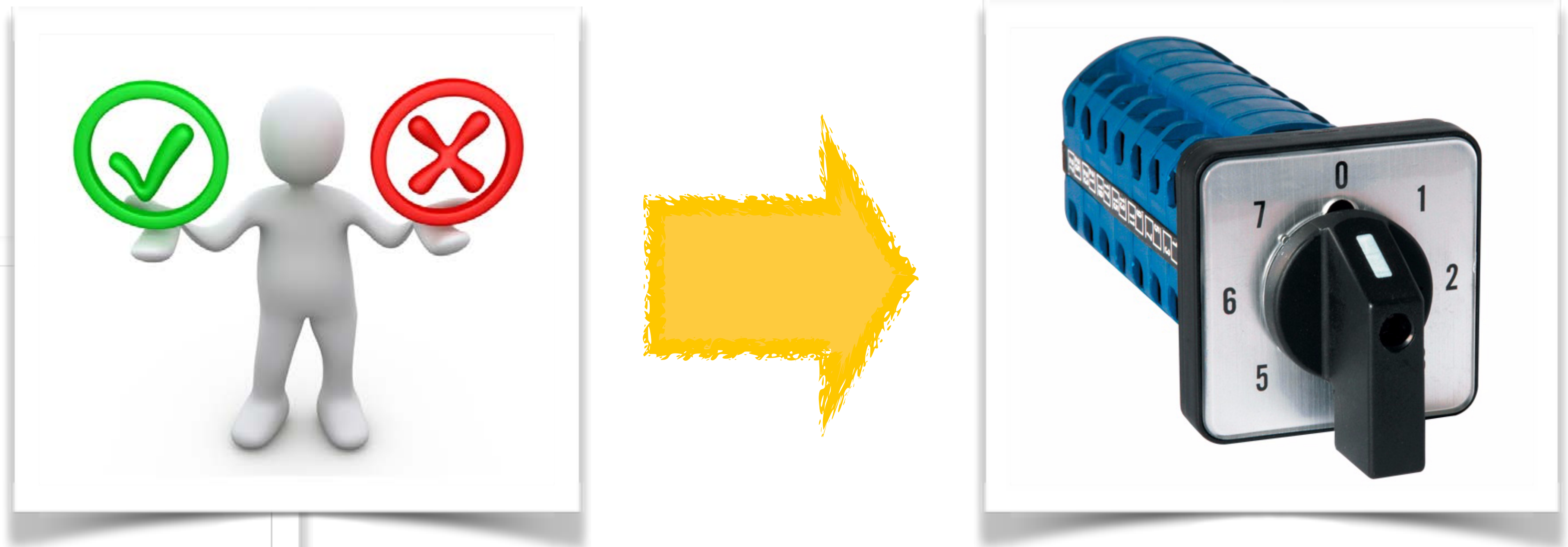
conditional branching

swift

```
func isLeap(year:Int) -> Bool {
     if year % 400 == 0 { return true }
     else if year % 100 == 0 { return false }
     else if year % 4 == 0 { return true }
     else  { return false }
}
```

```
func isLeap(year:Int) -> Bool {
     if (year % 4 == 0) && (year % 100 != 0) || (year % 400 == 0) { return true }
     else  { return false }
}
```

```
func isLeap(year:Int) -> Bool {
     return (year % 4 == 0) && (year % 100 != 0) || (year % 400 == 0)
}
```



conditional branching

switch / match

scala

```
i match {
  case 1  => println("January")
  case 2  => println("February")
  case 3  => println("March")
  ...
  case 12 => println("December")
  case whoa  => println("Unexpected: " + whoa.toString)
}
```

java

```
switch (n) {
    case 1: System.out.println("January"); break;
    case 2: System.out.println("February"); break;
    case 3: System.out.println("March"); break;
    ...
    case 12: System.out.println("December"); break;
    default: System.out.println("NOT A MONTH");
}
```

swift

```
let someCharacter: Character = "z"
switch someCharacter {
case "a":
  print("The first letter of the alphabet")
case "z":
  print("The last letter of the alphabet")
default:
  print("Some other character")
}
```

fallback case

reserved keywords

in a programming language, identifiers are lexical tokens chosen by the programmer to name various kinds of entities, e.g., variables, functions, types, etc.

in contrast, reserved keywords are words that cannot be chosen by the programmer to name entities and that has a predefined meaning, if, else, switch, etc.

command line arguments

scala

```
object HelloWorld extends App {
    if (args.length == 0) {
        println("Hello world")
    } else {
        println("Hello " + args(0))
   }
}
```

Args-Scala — -bash — ttys000

```
wallace-palace:Args-Scala garbi$ scalac HelloWorld.scala
wallace-palace:Args-Scala garbi$ scala HelloWorld
Hello world
wallace-palace:Args-Scala garbi$ scala HelloWorld Donald
Hello Donald
wallace-palace:Args-Scala garbi$
```

text input/output on the command line

when a program is launched on the command line, it can ask the user for text input and provide text output on the terminal

	input	output
python	`year = input("Give us a year: “)` `year = int(year)`	`print("Is {0} a leap year? {1}".format(year, isLeap(year)))`

scala	`import scala.io.StdIn.readLine`
	
	`val year = readLine("Choose a year: ").toInt`
		`print(s"Is $year a leap year? ${isLeap(year)}")`

java	`import java.util.Scanner;`
	
	`Scanner scanner = new Scanner(System.in);`
	`int year = scanner.nextInt();`
		`System.out.println("Is " + year + " a leap year? “ + isLeap(year));`

swift	`var year = Int(readLine()!)`	`print("Year \(year!) is leap: \(isLeap(year:year!))")`