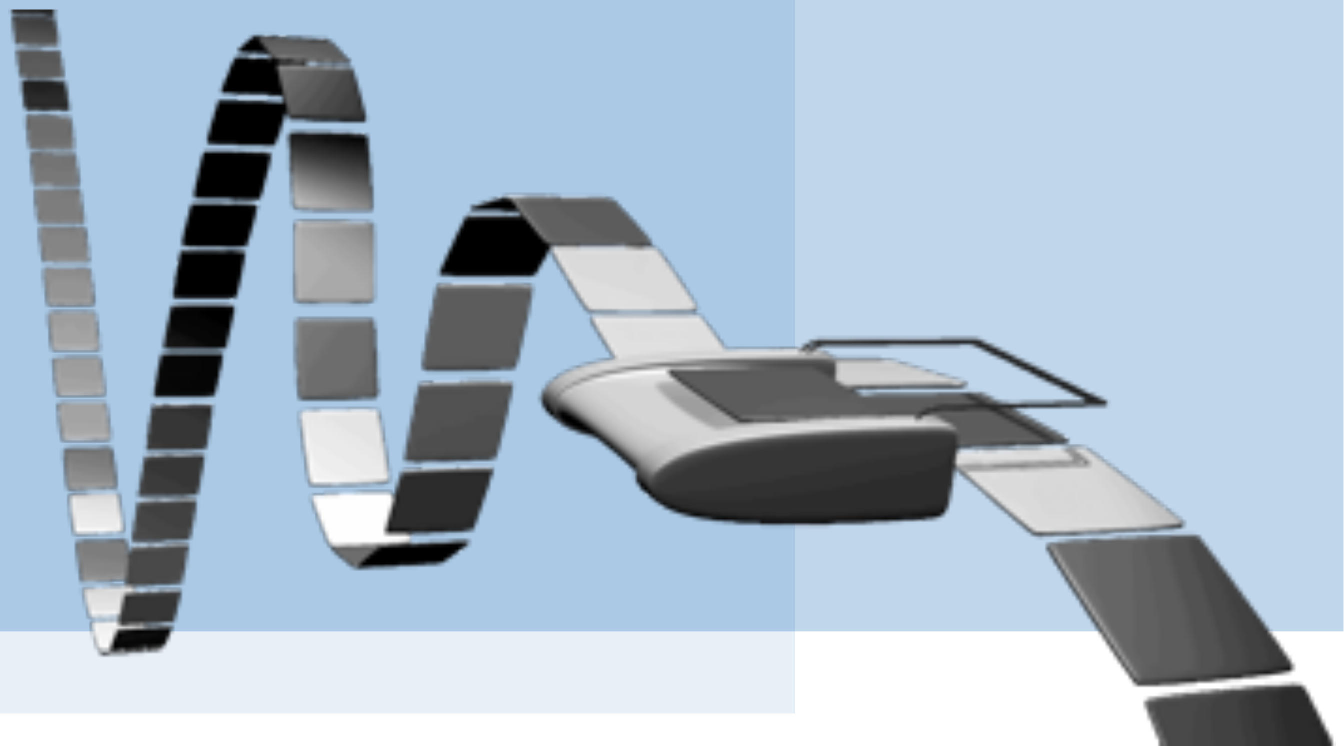
Einführung in die Informatik

Programming Languages

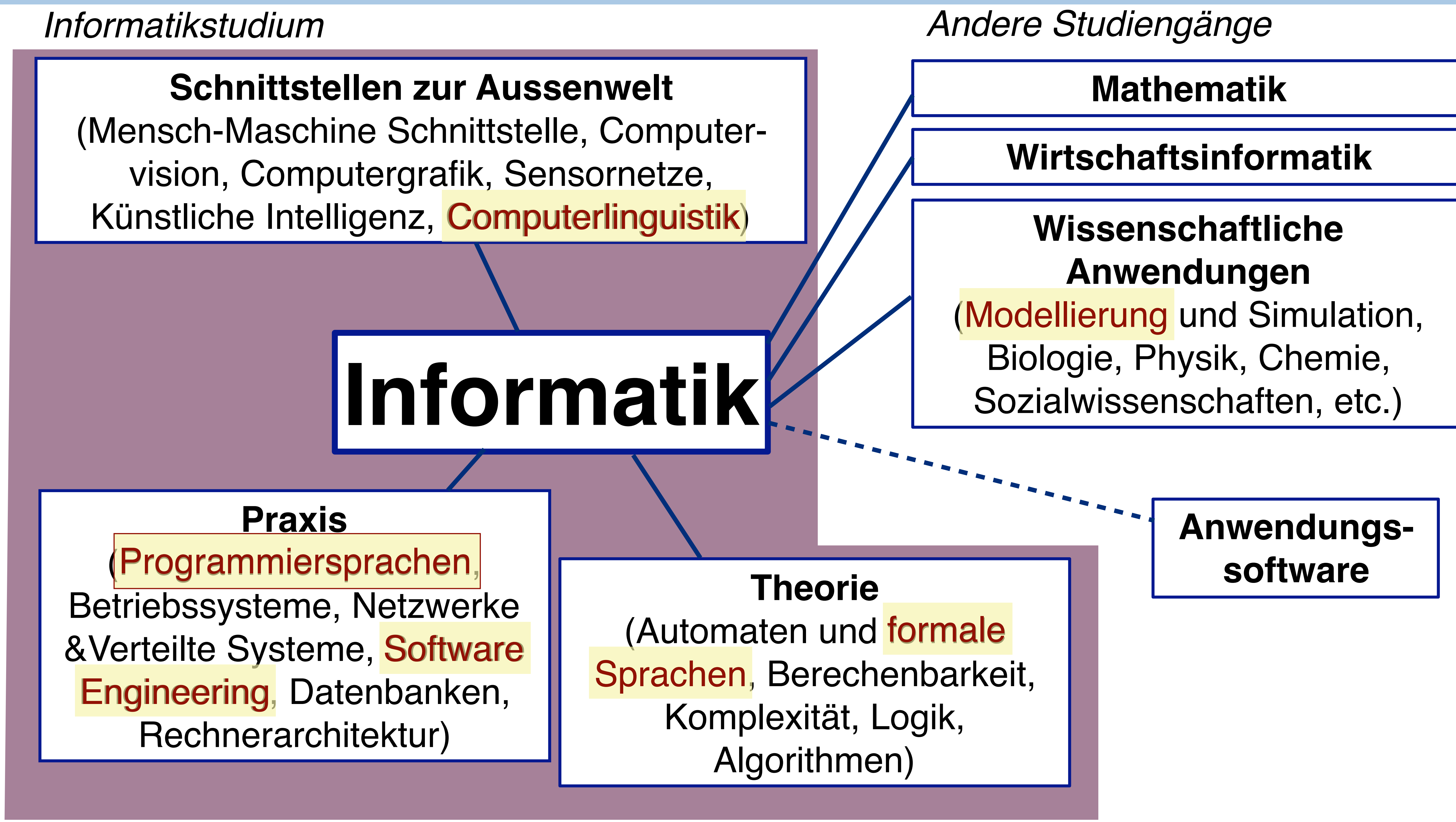
Prof. O. Nierstrasz



Roadmap

> What is a programming language?
> Historical Highlights
> Conclusions

Übersicht


Informatikstudium
Andere Studiengänge
Schnittstellen zur Aussenwelt (Mensch-Maschine Schnittstelle, Computer-vision, Computergrafik, Sensornetze, Künstliche Intelligenz, Computerlinguistik)
Informatik
Mathematik
Wirtschaftsinformatik
Wissenschaftliche Anwendungen (Modellierung und Simulation, Biologie, Physik, Chemie, Sozialwissenschaften, etc.)
Anwendungs-software
Praxis (Programmiersprachen, Betriebssysteme, Netzwerke &Verteilte Systeme, Software Engineering, Datenbanken, Rechnerarchitektur)
Theorie (Automaten und formale Sprachen, Berechenbarkeit, Komplexität, Logik, Algorithmen)

Roadmap

> **What is a programming language?**
> Historical Highlights
> Conclusions

What is a language?

Jack and Jill went up the hill …

Language = a set of *sequences* of symbols that we *interpret* to attribute *meaning*

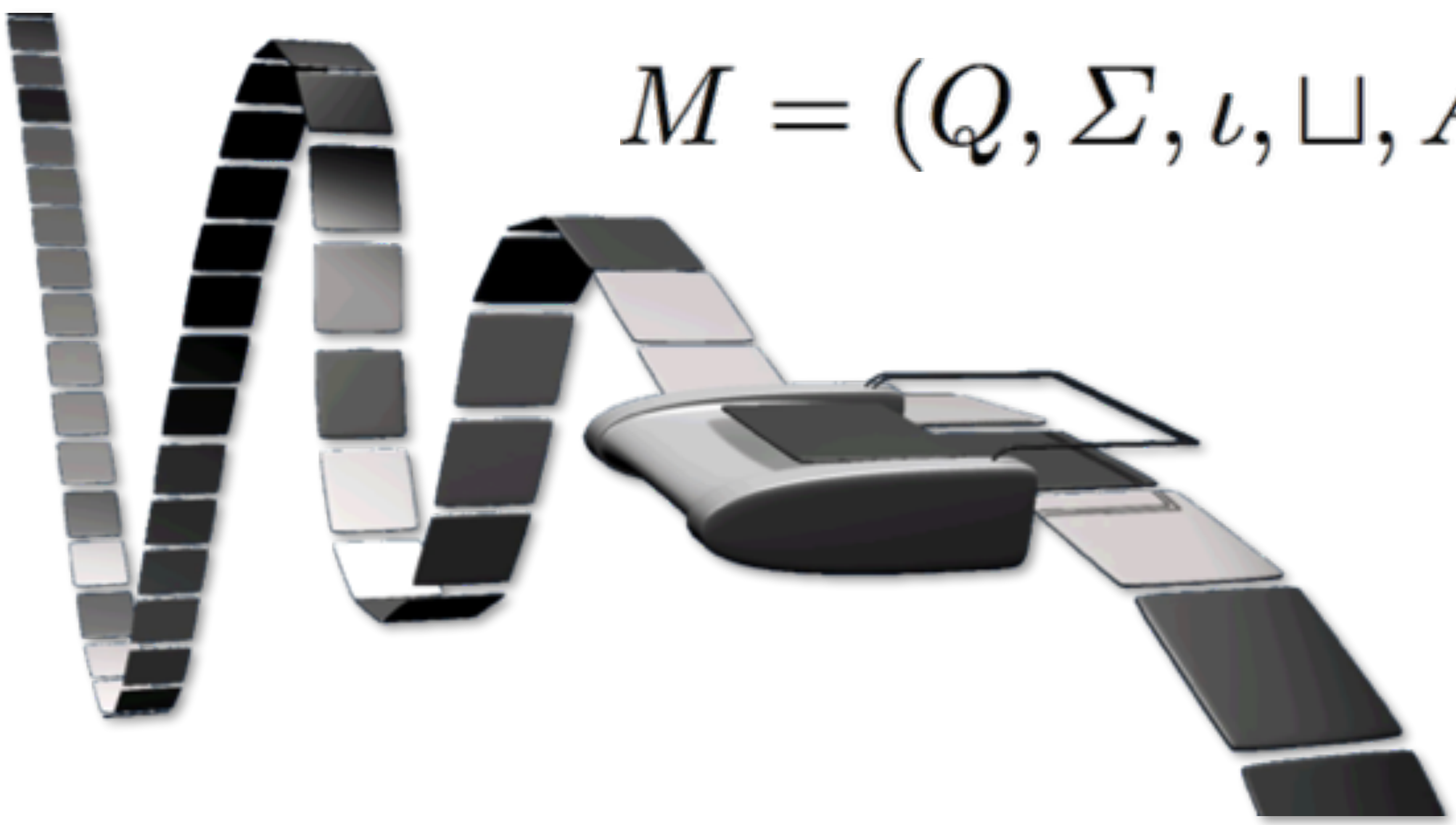
Languages consist of sets of (spoken or written) phrases that have meaning for us.

There are many natural languages, and all of them have similar concepts: cases, tenses, nouns, verbs. The number of cases differ, as well as the grammatical rules.

So is with programming languages: there are many, but there are much fewer concepts that all of them employ and even fewer that represent variation points.

What is a formal language?

A *Turing machine* reads (and writes) a tape of 0s and 1s



$$M = (Q, \Sigma, \iota, \sqcup, A, \delta)$$

The *language* it accepts is the set of strings that leave it in an *accepting state*

The *language* of a Turing machine is the set of inputs that it accepts.

- Q is a finite set of *states* (of the machine)
- ∑ is a finite set of *tape symbols*
- i is an *initial state* (of the machine)
- _ is a *blank* symbol
- ∂ is a *transition function* (state and tape symbol → new state, symbol to write, move left or right)
- A is set of *accepting states* (stop if we reach one of these states)

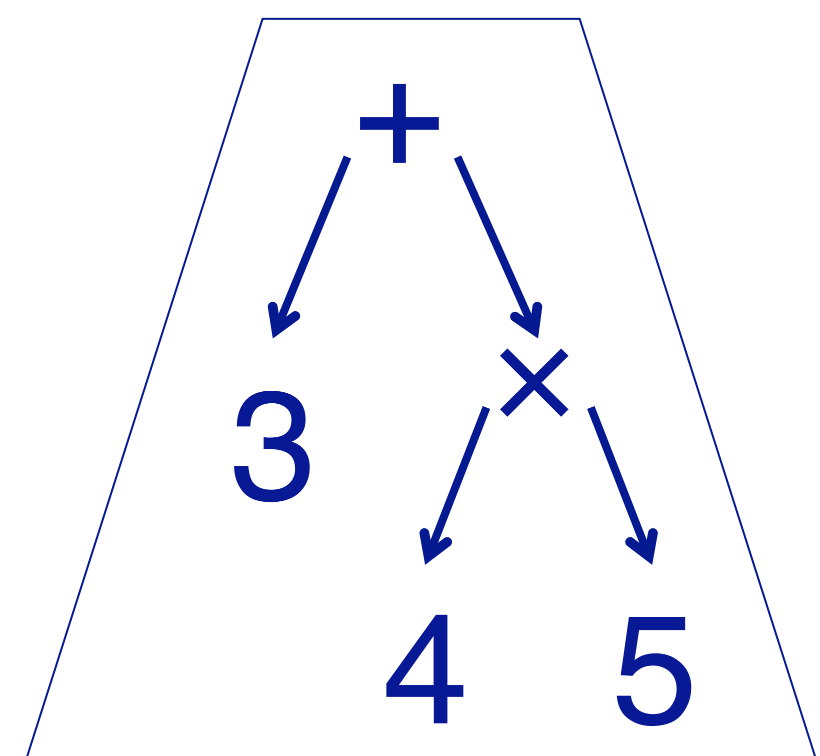
Input is a tape with a *finite set* of non-blank symbols.

How can we describe formal languages?

Use a set of *rules* ($\alpha \rightarrow \beta$) to describe the *structure* of the language

expression → number
expression → expression + expression
expression → expression × expression
number → digit
number → digit number

3 + 4 × 5



3 + 4 × × 5 ⟶ error!

Rules are used to *recognize* a particular string of symbols as having a particular structure. If the rules cannot recognize the string, then it is not in the language.

Take for example, the grammar for mathematical expressions presented on the slide. These rules will recognize "3 + 4 × 5" as a valid string in the language of arithmetic expressions, but will reject "3 + 4 × × 5"

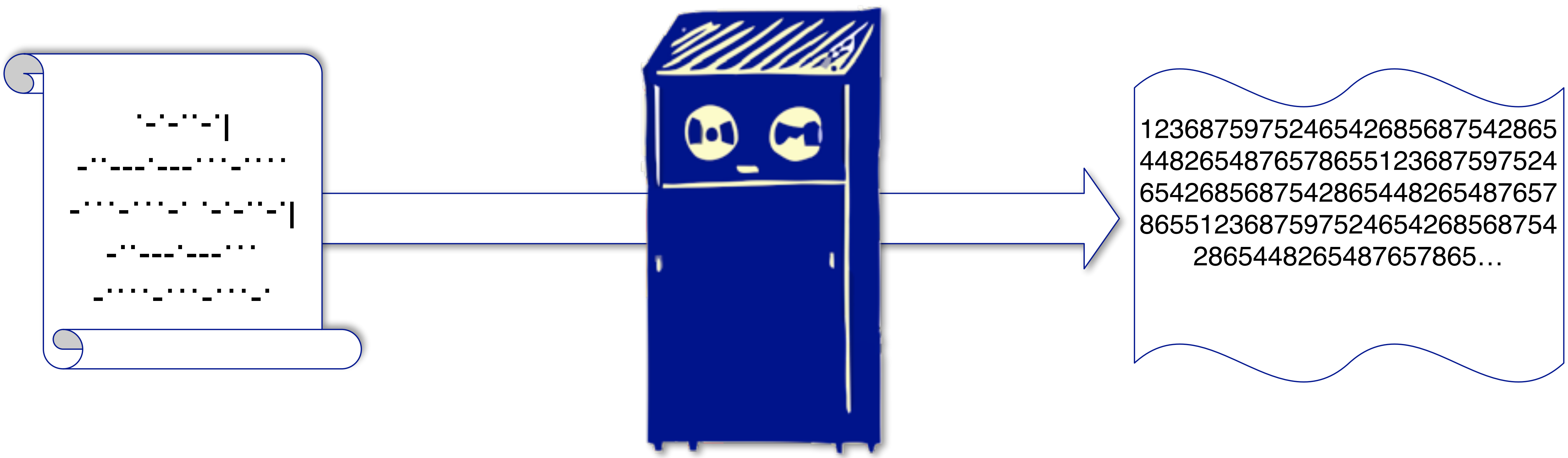
Aside: Different constraints on the rules we have give us *different kinds of languages*, known as the *Chomsky hierarchy*:

- type 0 grammars are *unrestricted* — they are equivalent to Turing machines
- type 1 grammars are *context sensitive*: aAb → agb — equivalent to linear-bounded non-deterministic Turing machines (!)
- type 2 grammars are *context-free*: A → g — equivalent to pushdown automata
- type 3 grammars are *regular*: A → a or A → aB — equivalent to finite state automata (or regular expressions)

https://en.wikipedia.org/wiki/Chomsky_hierarchy

What is a Programming Language? (take 1)

A language to instruct a computer to compute “stuff” …



But how does the computer interpret language?

(1) A PL is simply a language for communicating instructions to the computer.

It was not always like that: back in the day programmers used to flip switches to program a computer. Nowadays however, we write programs in “high-level languages”. These programs eventually make the computer do stuff.

But how do we bridge the gap between the programs we write and the computer executing?

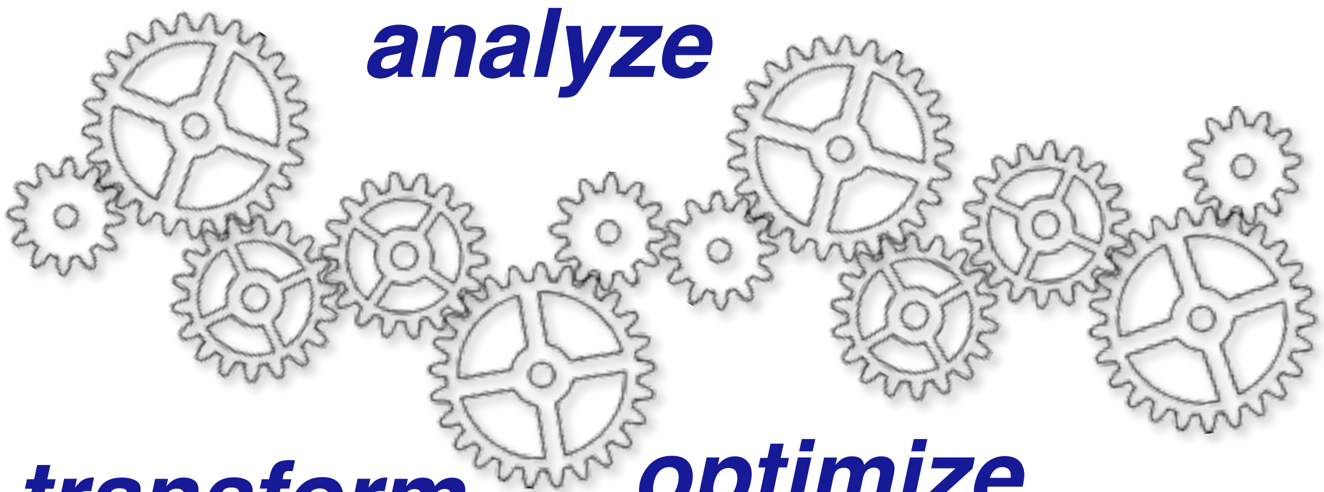
What is a Programming Language? (take 2)

What the compiler will handle …

.-…-…-….-
…---.---..-
l.-..-.-. .-…-…-
….-…---.---..-
l.-..-.-.

parse

analyze



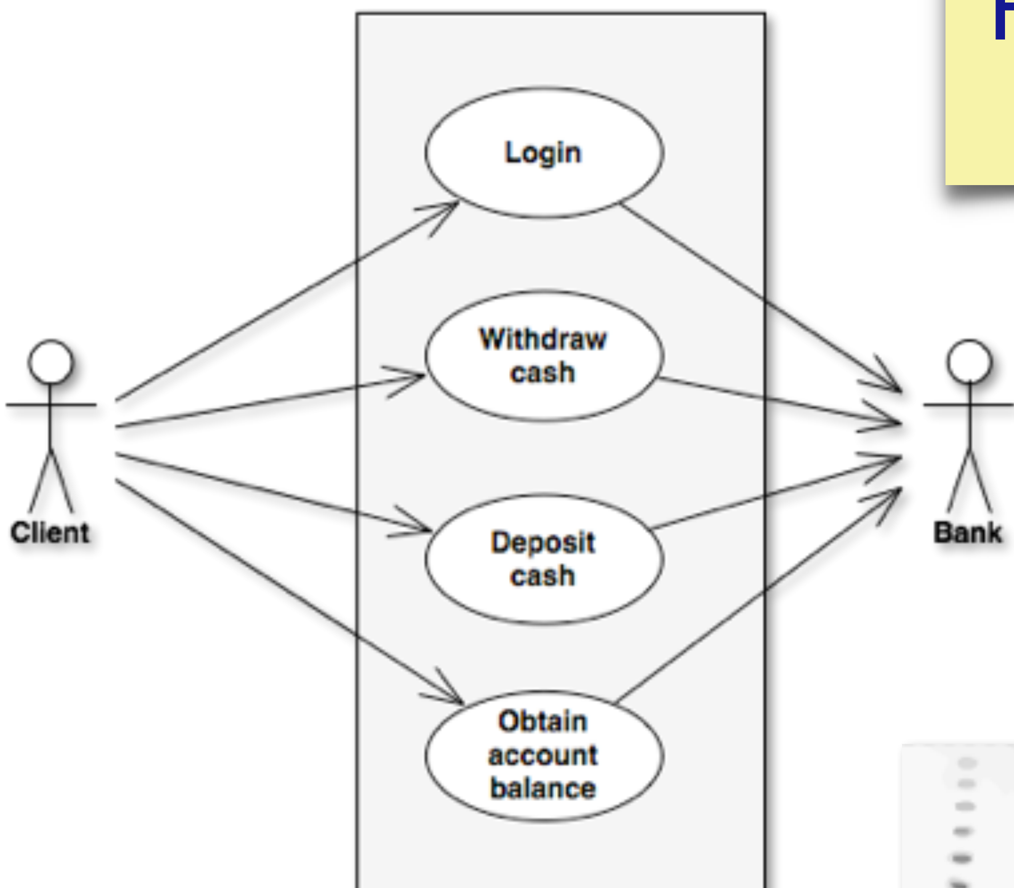
transform ***optimize*** ***generate***

0100100101100
11001000111010
1011010000110
10101010010111
10011111000101
01000101011…

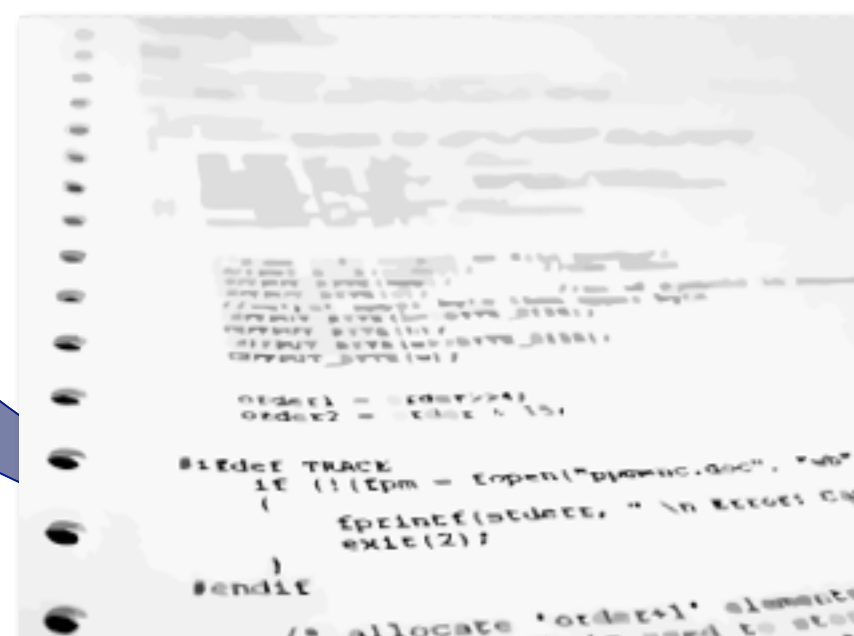
But what about the programmer?

A PL comes with a *compiler* that translates the programs written according to the rules down into the machine language. Alternatively there may be an *interpreter* that directly interprets the code without generating a machine executable.

What is a Programming Language? (take 3)



Programming is *modeling*



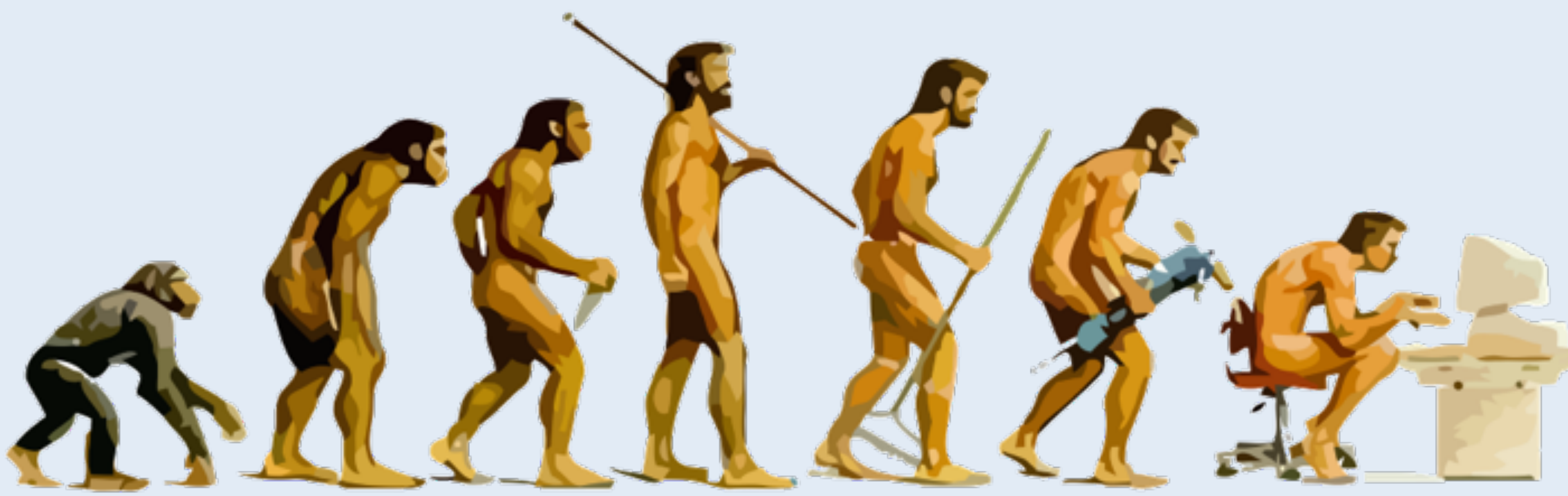
The take-home message is that *programming is modeling*.

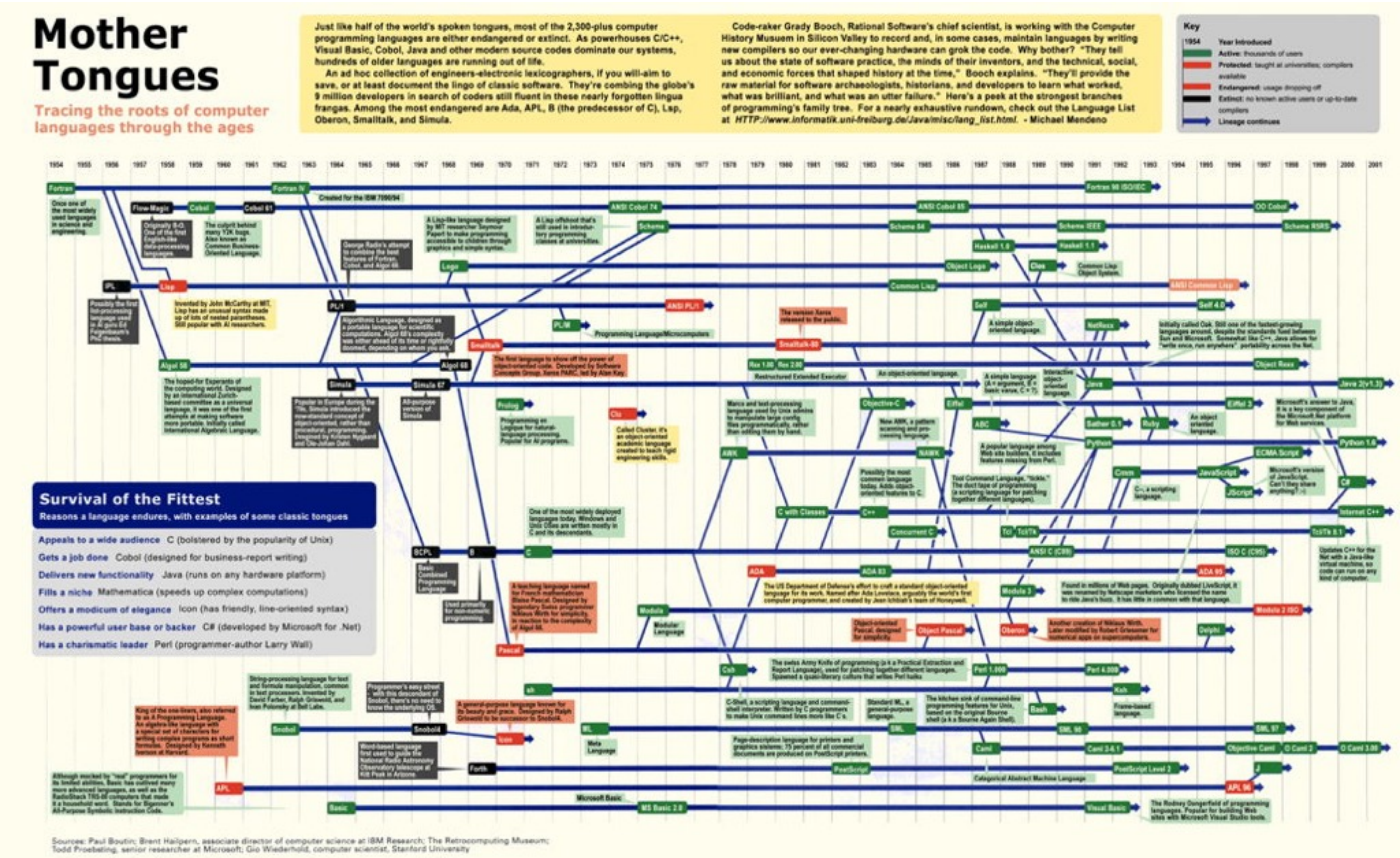
Programs are *executable models* that are used to achieve some effect in the real world. With a good design, the program code reflects clearly the models as we want them to be.

Programming languages offer us a variety of different tools for expressing executable models. If we pick the right tool, the job is easier.

Roadmap

- What is a programming language?
- **Historical Highlights**
- Conclusions




Mother Tongues
Tracing the roots of computer languages through the ages
Just like half of the world's spoken tongues, most of the 2,300-plus computer programming languages are either endangered or extinct. As powerhouses C/C++, Visual Basic, Cobol, Java and other modern source codes dominate our systems, hundreds of older languages are running out of life.
An ad hoc collection of engineers-electronic lexicographers, if you will-aim to save, or at least document the lingo of classic software. They're combing the globe's 9 million developers in search of coders still fluent in these nearly forgotten lingua frangas. Among the most endangered are Ada, APL, B (the predecessor of C), Lisp, Oberon, Smalltalk, and Simula.
Code-raker Grady Booch, Rational Software's chief scientist, is working with the Computer History Musuem in Silicon Valley to record and, in some cases, maintain languages by writing new compilers so our ever-changing hardware can grok the code. Why bother? "They tell us about the state of software practice, the minds of their inventors, and the technical, social, and economic forces that shaped history at the time," Booch explains. "They'll provide the raw material for software archaeologists, historians, and developers to learn what worked, what was brilliant, and what was an utter failure." Here's a peek at the strongest branches of programming's family tree. For a nearly exhaustive rundown, check out the Language List at HTTP://www.informatik.uni-freiburg.de/Java/misc/lang_list.html. - Michael Mendeno
Key
1954 Year Introduced
Active: thousands of users
Protected: taught at universities; compilers available
Endangered: usage dropping off
Extinct: no known active users or up-to-date compilers
Lineage continues
Fortran
Fortran IV
Fortran 90 ISO/IEC
Cobol
ANSI Cobol 74
ANSI Cobol 85
OO Cobol
Scheme
Lisp
Common Lisp
PL/I
Algol 58
Algol 68
Simula
Simula 67
Smalltalk
Java
C
C++
ADA
Pascal
Basic
Visual Basic
Survival of the Fittest
Reasons a language endures, with examples of some classic tongues
Appeals to a wide audience C (bolstered by the popularity of Unix)
Gets a job done Cobol (designed for business-report writing)
Delivers new functionality Java (runs on any hardware platform)
Fills a niche Mathematica (speeds up complex computations)
Offers a modicum of elegance Icon (has friendly, line-oriented syntax)
Has a powerful user base or backer C# (developed by Microsoft for .Net)
Has a charismatic leader Perl (programmer-author Larry Wall)
Sources: Paul Boutin; Brent Hailpern, associate director of computer science at IBM Research; The Retrocomputing Museum; Todd Proebsting, senior researcher at Microsoft; Gio Wiederhold, computer scientist, Stanford University

Why so many?!

The graphic shows a small extract of the family tree of programming languages:

http://visual.ly/mother-tongues-—-tracing-roots-computer-languages-through-ages

Thousands of PLs and dialects have been invented over the years, and the number continues to grow.

Why are there so many programming languages?

A PL is a tool. Tools should fit the task at hand. Since tasks change, new tools continue to be invented …

What do programming languages have in common?

comments

functions

variables

keywords

numbers, strings

control constructs

expressions

statements

```ruby
# Compute factorials
def fact(n)
  if n == 0
    1
  else
    n * fact(n-1)
  end
end

puts fact(ARGV[0].to_i)
```

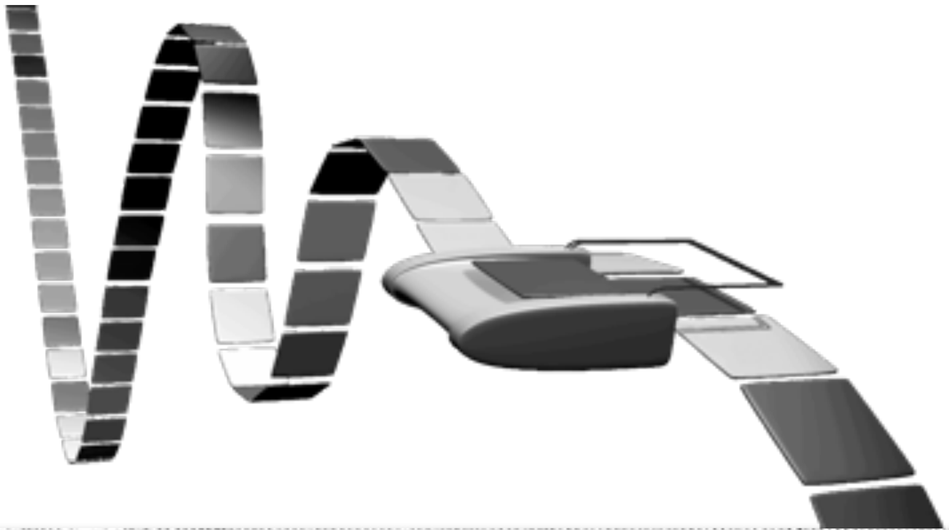
A fragment of Ruby code

All PLs have certain features in common. This Ruby fragment shows many of the basic constructs.

Note that technically n is not just a variable but an argument to a function. Also in Ruby, every statement is actually an expression, returning a result.

Expressive power

Formally, all programming languages are equivalent …



So what? …

Nearly all programming languages have the expressive power of Turing machines (or, equivalently, the Lambda calculus).

But they are not equally good at expressing solutions to different kinds of problems …

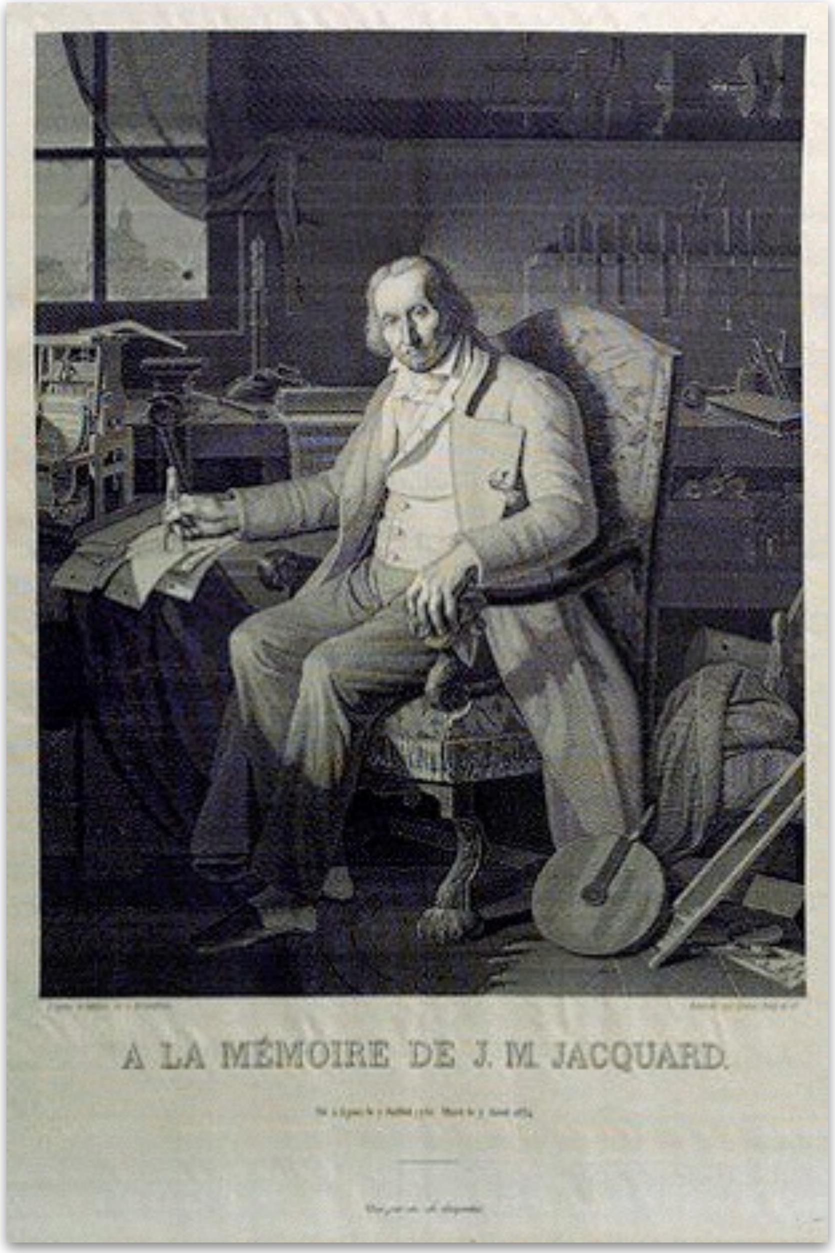
The term “Turing tarpit” itself comes from the 1982 paper “Epigrams on Programming” by Alan Perlis:

> *Beware of the Turing tar-pit in which everything is possible but nothing of interest is easy.*

http://pu.inf.uni-tuebingen.de/users/klaeren/epigrams.html

“tar pit” = “Teergrube”

Jacquard loom — 1801



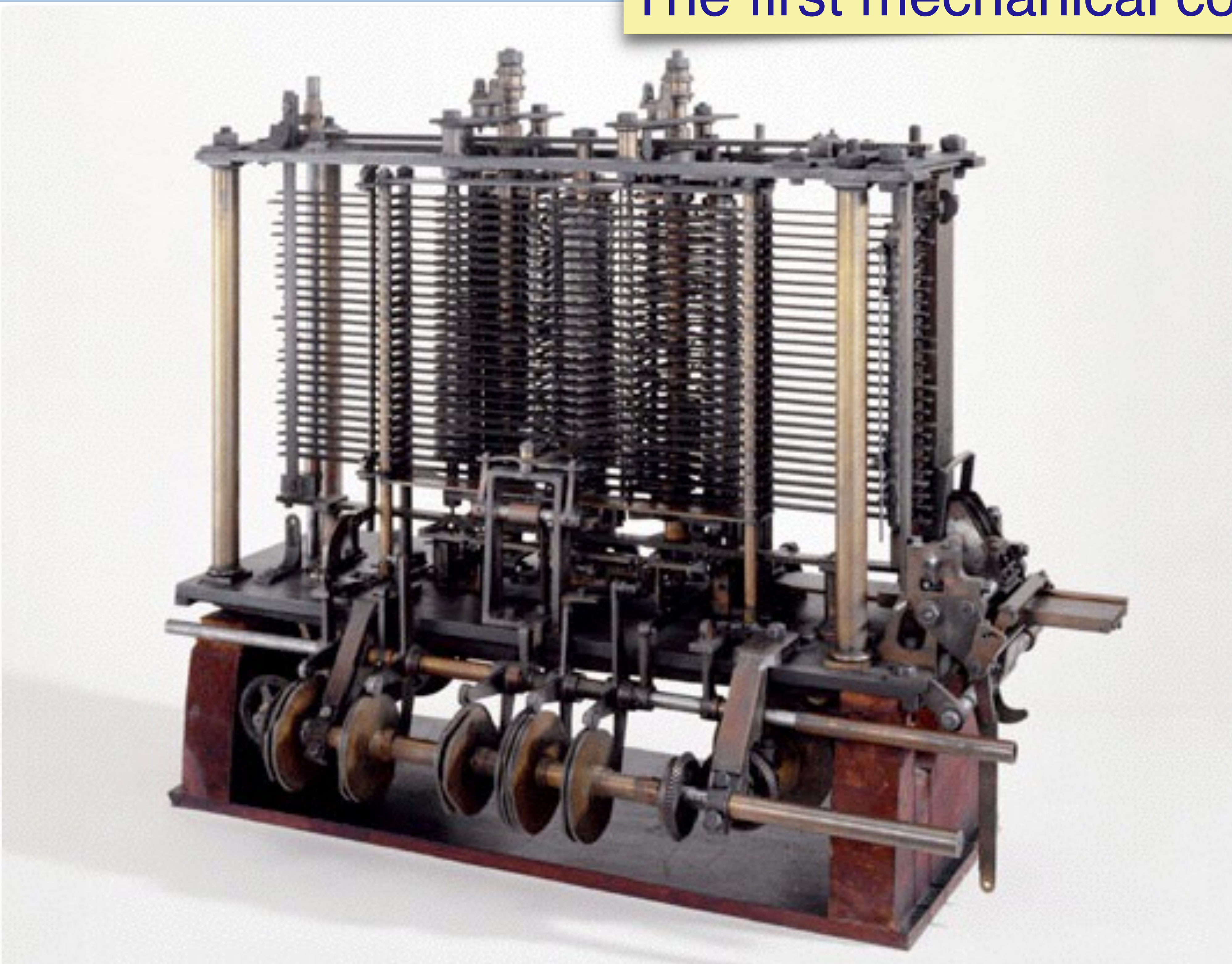
Punch cards are invented

Holes in punched cards controlled the way the loom’s arm moved to produce different decorative patterns.

The image to the right was generated on a Jacquard loom, by using 24,000 punched cards to create.

Babbage's Analytical Engine — 1822

The first mechanical computer

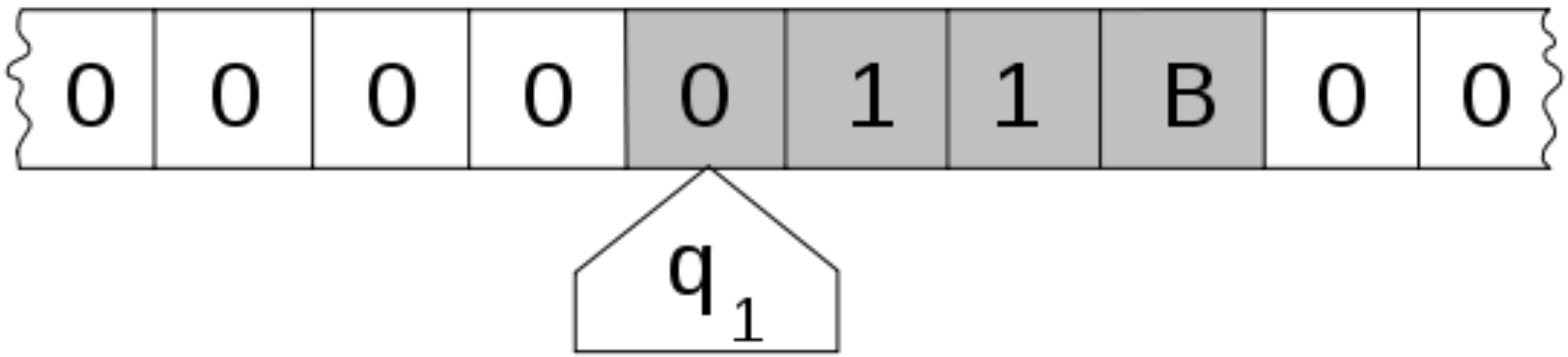


Charles Babbage, english scientist, 1791-1871, was the first to design a computing machine to solve polynomials: the *Difference Engine*.

http://en.wikipedia.org/wiki/Charles_Babbage

Afterwards he designed the *Analytical engine*, the first computing machine capable of general-purpose computations, not only polynomials. The Analytical Engine is the reason for which he is considered a pioneer of computing science.

Turing machine — 1936



The first abstract model of a computer

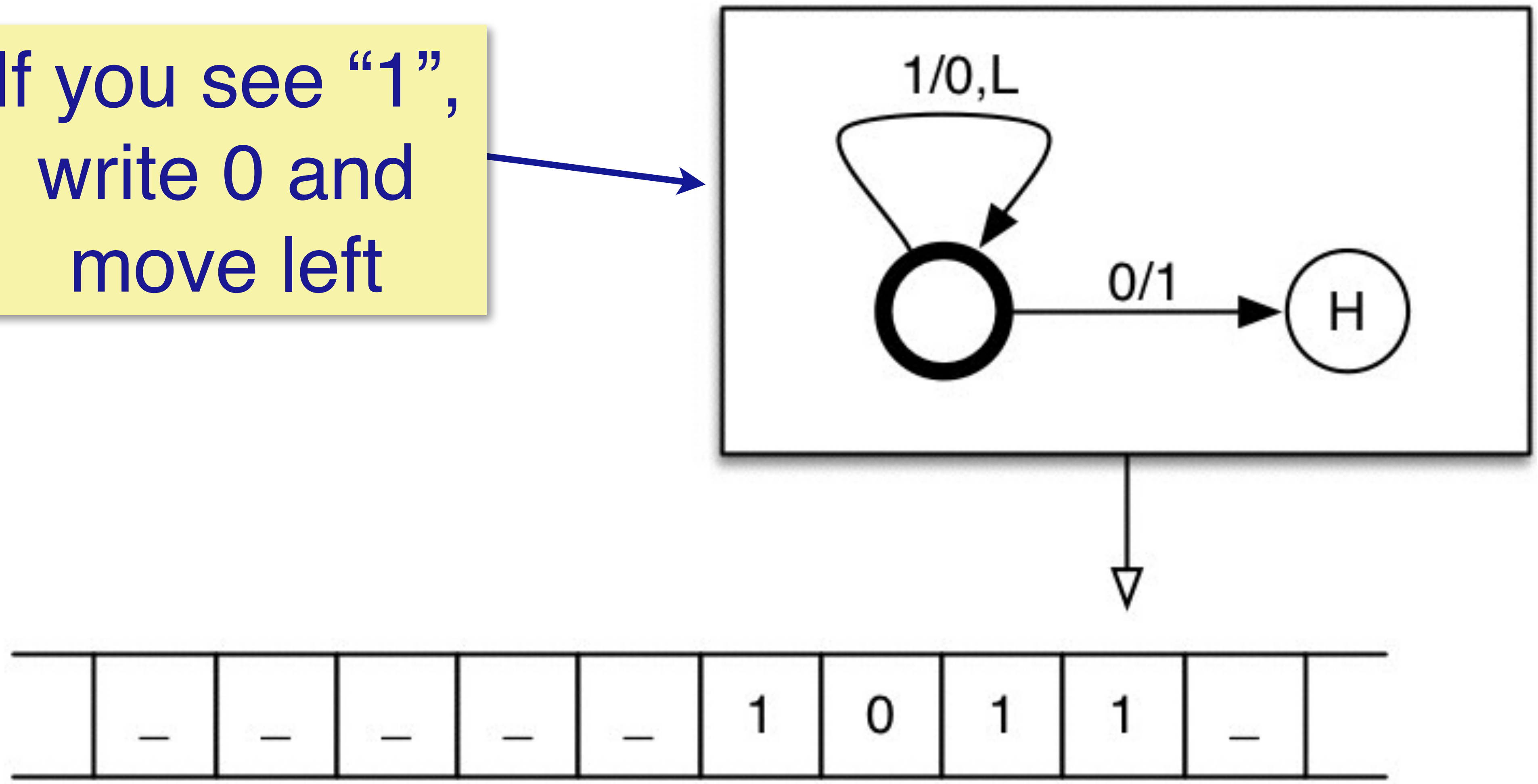
Alan Turing, a British mathematician, wrote the article on the Turing machine when he was 24.

http://en.wikipedia.org/wiki/Alan_Turing

A finite state machine reads and writes an infinite tape (with a finite set of symbols). It is undecidable whether a given machine and input will terminate.

An incrementing TM (1)

If you see “1”, write 0 and move left



Binary “1011” = Decimal “11”

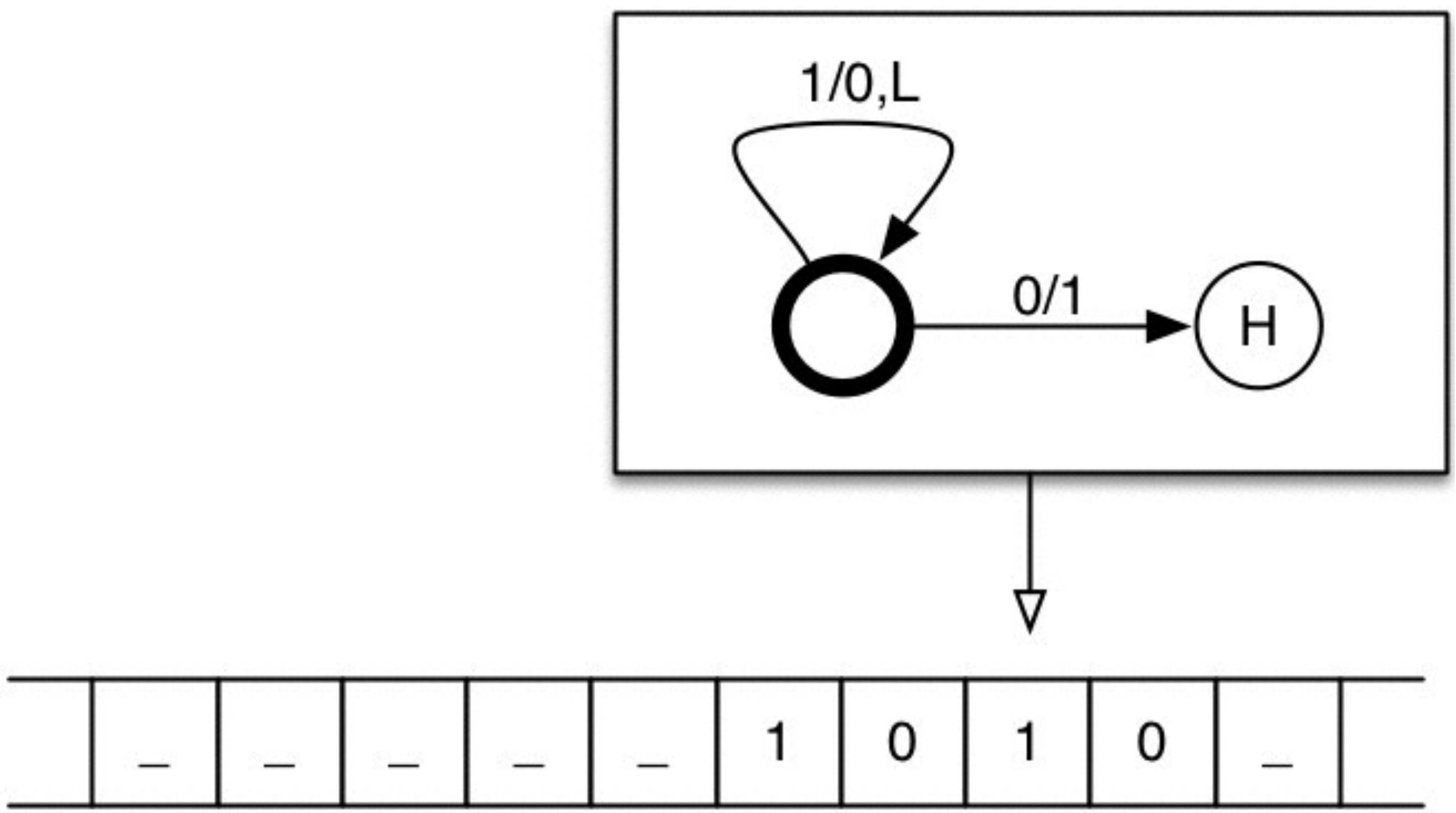
A Turing machine consists of a Finite State Automaton (FSA) and a tape with a finite number of non-blank symbols. The FSA is always positioned at location on the tape. Each execution step consists of:

1. reading the current symbol
2. possibly writing a symbol at the current position
3. possibly moving left or right
4. possibly terminating ("halt" or "accept")

This machine will add 1 to the binary number on the tape.

In the first step it will read 1, write 0, move left, and transition to the initial (current) state.

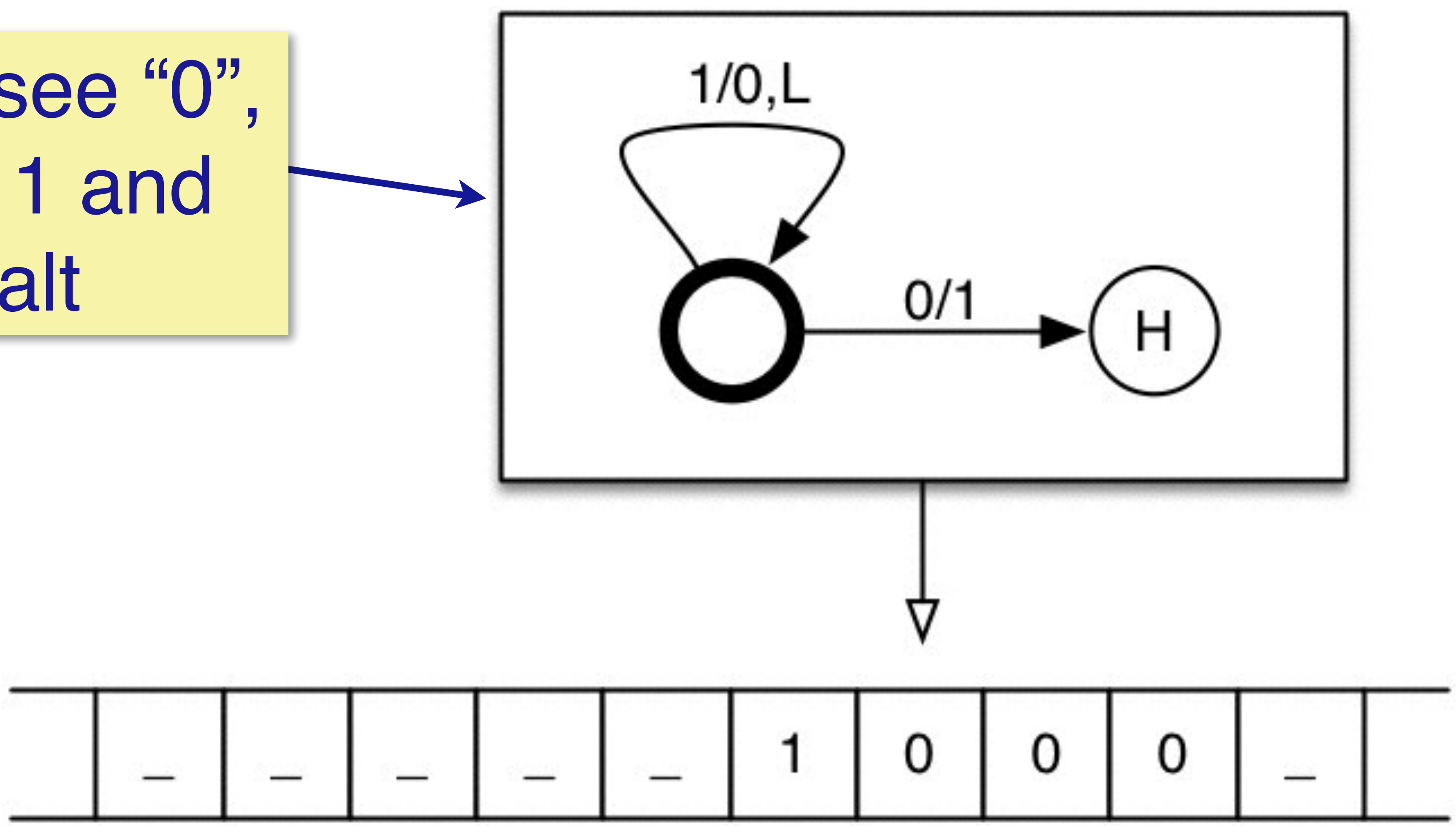
An incrementing TM (2)



Again, read 1, write 0, move left, transition to initial state.

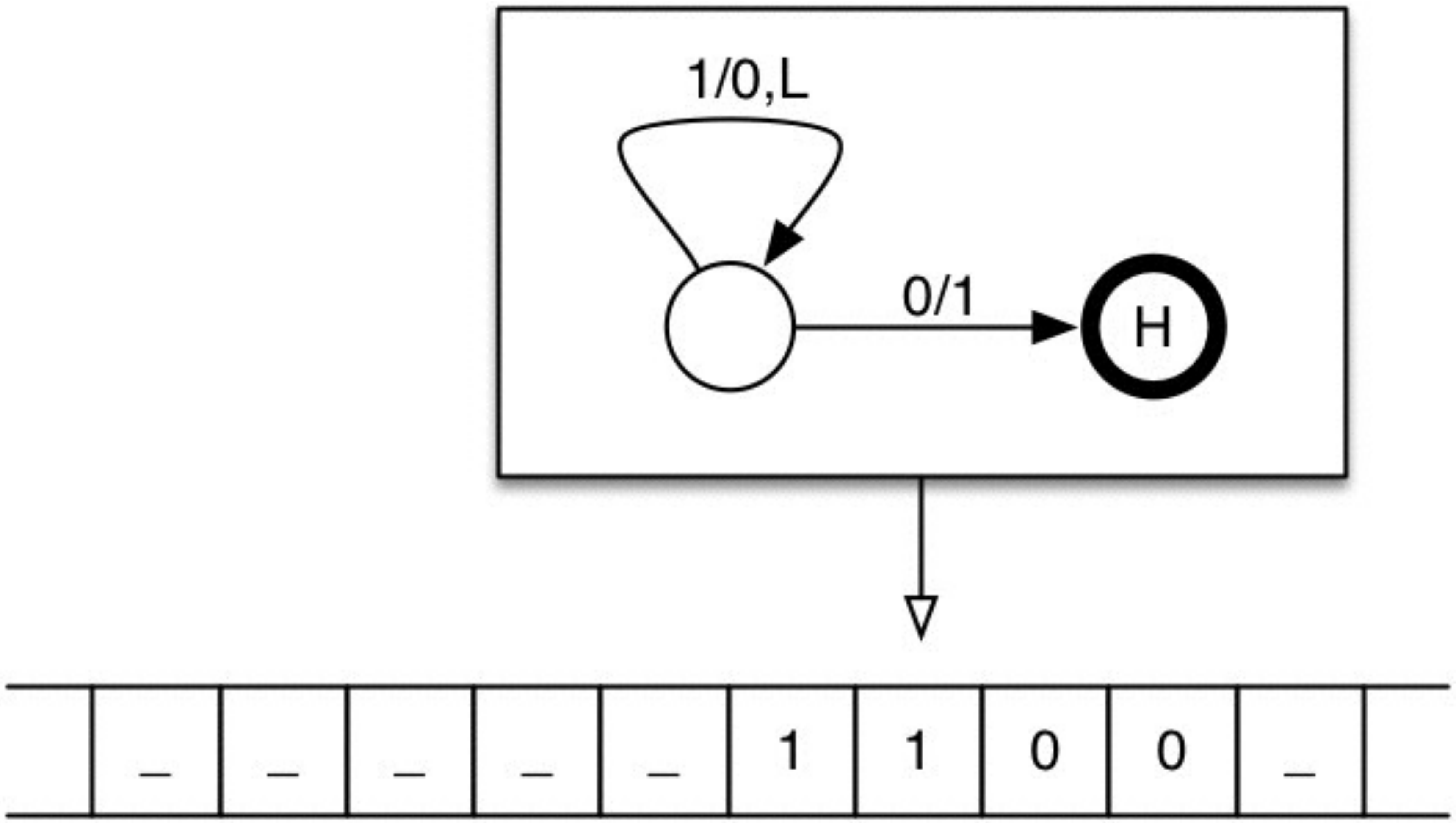
An incrementing TM (3)

If you see "0", write 1 and halt



Read 0, write 1, transition to accepting state (H).

An incrementing TM (4)



Binary “1100” = Decimal “12”

ENIAC — 1946

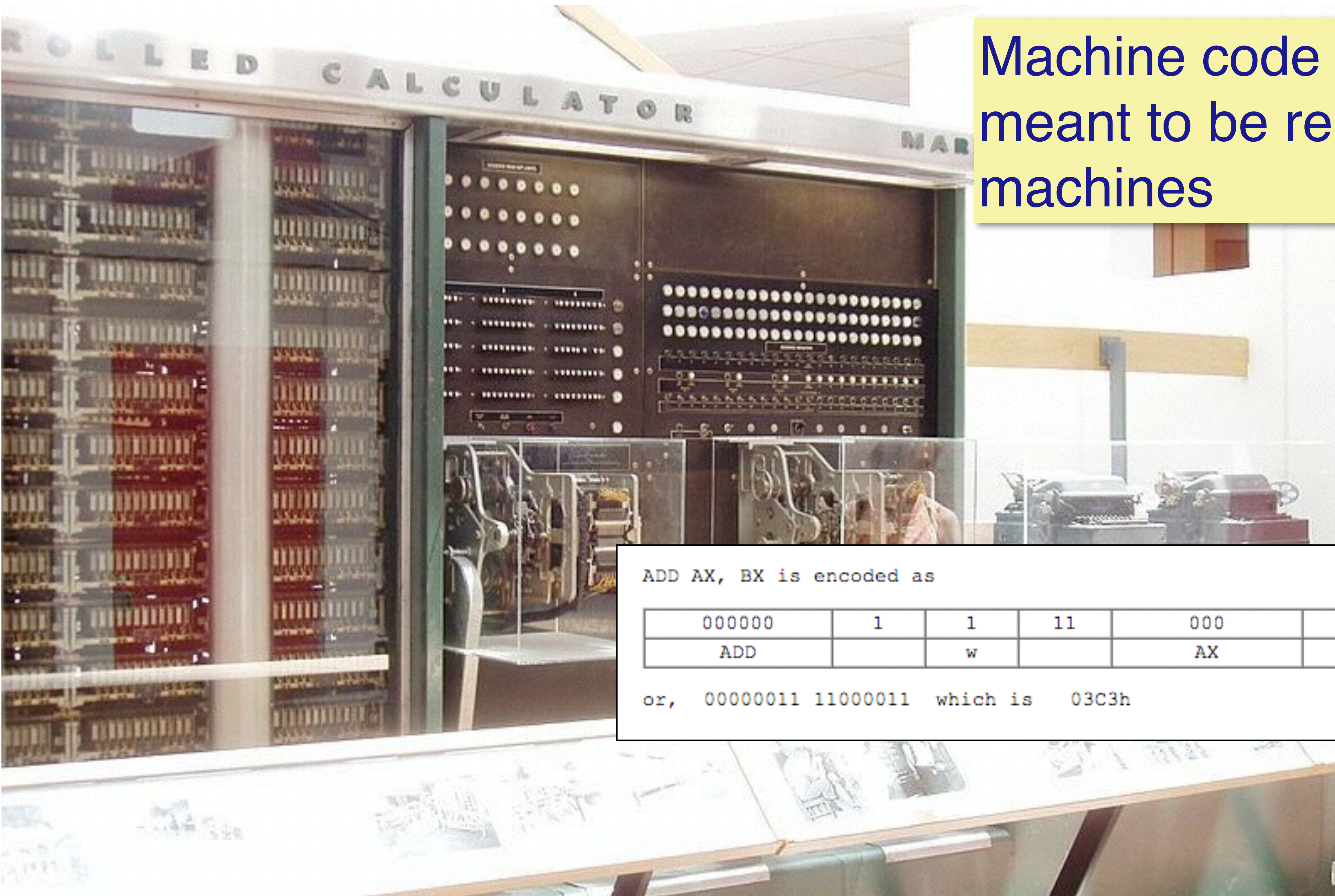
Programming = reconfiguring the computer

The ENIAC was the first electronic general-purpose computer, mostly used for ballistic computations in the military.

Once the program was written on paper, it took several people several days to program the computer by setting switches and plugging in cables!

- https://en.wikipedia.org/wiki/ENIAC
- https://www.youtube.com/watch?v=goi6NAHMKog

1st generation: Machine code — 1944



Machine code is only meant to be read by … machines

ADD AX, BX is encoded as

000000	1	1	11	000	011
ADD		w		AX	BX

or, 00000011 11000011 which is 03C3h

The Harvard Mark 1 was programmed with machine code on punched paper tape. The were no loops (iteration instructions).

https://en.wikipedia.org/wiki/Harvard_Mark_I

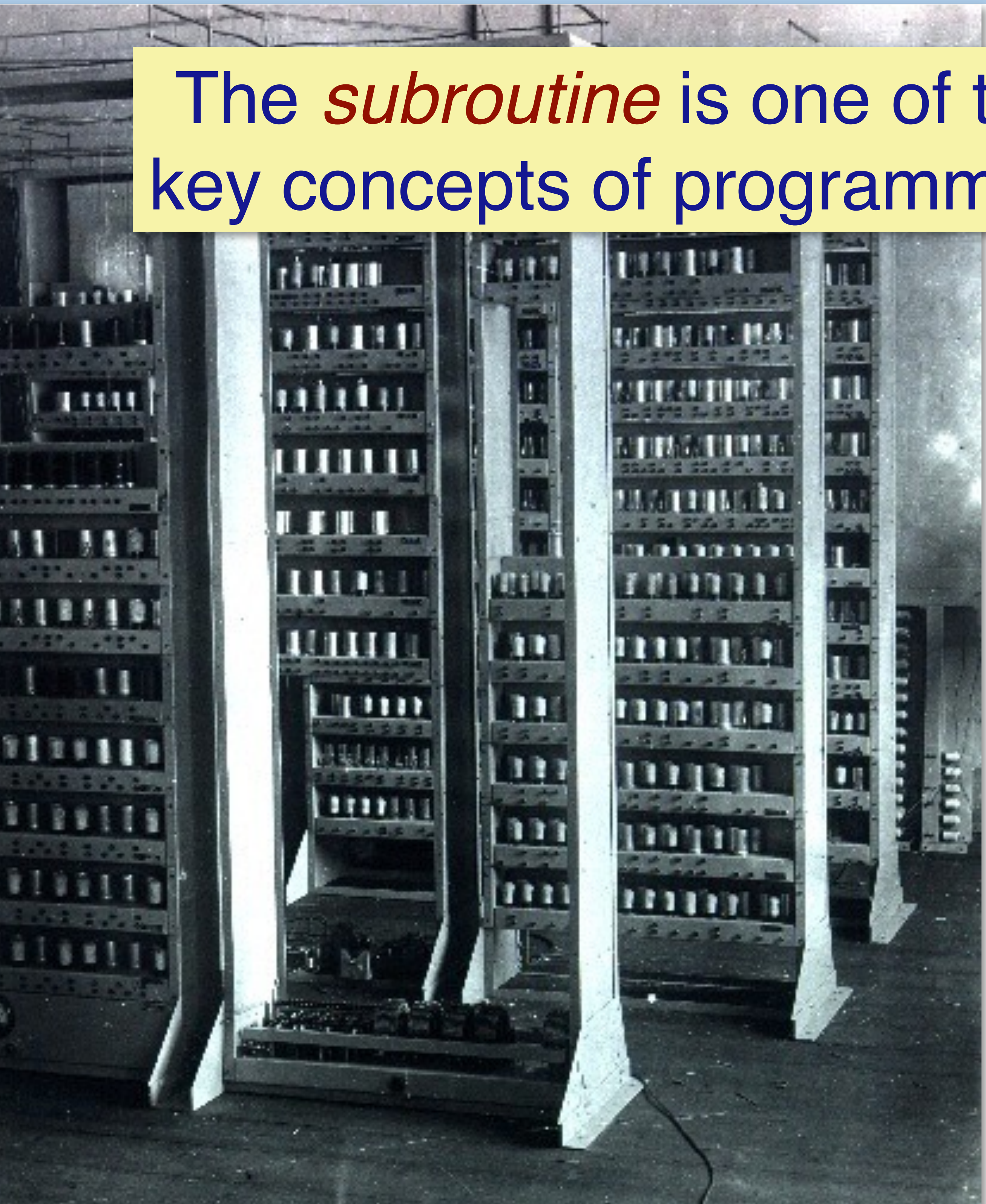
http://www.computerhistory.org/timeline/?category=cmptr

Machine code was really written only for the machine to understand. And the computer does not understand much: it merely understands whether a bit is on or not (i.e., whether there is tension in a circuit or not).

Each instruction is usually very simple and does not do much. The instruction in the example is adds two special regions of memory.

In order to be able to be able to write machine code, one must understand the architecture of the machine. And there are many types of machines, with different numbers of registers, and different instructions.

Subroutines — 1949



The *subroutine* is one of the key concepts of programming

```
CODE
...

    mov     r0, #1
    mov     r1, #3
    mov     r2, #-4
    bl      do_something
    mov     r1, r0
    bl      printf
    mov     r0, #0
```

In the code highlighted the BL instruction calls the do_something subroutine. Before the introduction of the subroutine concept, writing programs was much harder as one had to always keep track exactly of the places where jump instructions had to be executed.

David Wheeler is credited with the invention of the “closed subroutine”. Dijkstra points to this as one of the most fundamental contributions to PL design. Before subroutines sub-calculations were implemented with Jump instructions and conditionals.

http://en.wikipedia.org/wiki/Subroutine

http://en.wikipedia.org/wiki/David_Wheeler_(computer_scientist)

2nd generation: assembler — early 1950s

Assembly code introduces *symbolic names* (for humans!)

Address	Label	Instruction (AT&T syntax)	Object code[16]
		.begin	
		.org 2048	
	a_start	.equ 3000	
2048		ld length,%	
2064		be done	00000010 10000000 00000000 00000110
2068		addcc %r1,-4,%r1	10000010 10000000 01111111 11111100
2072		addcc %r1,%r2,%r4	10001000 10000000 01000000 00000010
2076		ld %r4,%r5	11001010 00000001 00000000 00000000
2080		ba loop	00010000 10111111 11111111 11111011
2084		addcc %r3,%r5,%r3	10000110 10000000 11000000 00000101
2088	done:	jmpl %r15+4,%r0	10000001 11000011 11100000 00000100
2092	length:	20	00000000 00000000 00000000 00010100
2096	address:	a_start	00000000 00000000 00001011 10111000
		.org a_start	
3000	a:		

The EDSAC computer (Britain, 1949) was the first computer to use an assembly language. Such languages introduce symbolic names for variables, jump locations etc.

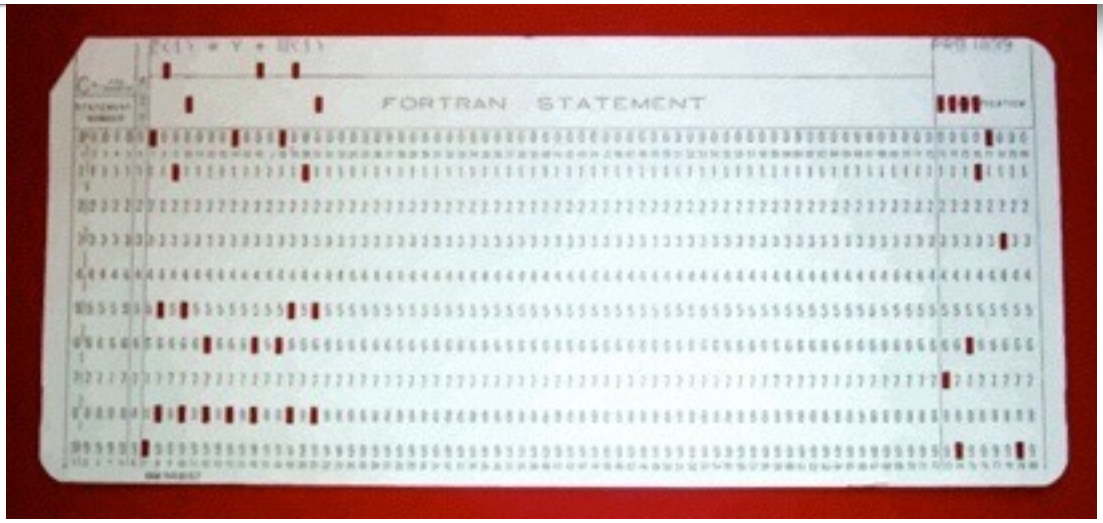
http://en.wikipedia.org/wiki/Assembly_language#Historical_perspective

http://en.wikipedia.org/wiki/Electronic_delay_storage_automatic_calculator

3rd generation: FORTRAN — 1955

High-level languages are born

```
C AREA OF A TRIANGLE - HERON'S FORMULA
C INPUT - CARD READER UNIT 5, INTEGER INPUT
C OUTPUT - LINE PRINTER UNIT 6, REAL OUTPUT
C INPUT ERROR DISPAY ERROR OUTPUT CODE 1 IN JOB CONTROL LISTING
      INTEGER A,B,C
      READ(5,501) A,B,C
  501 FORMAT(3I5)
      IF(A.EQ.0 .OR. B.EQ.0 .OR. C.EQ.0) STOP 1
      S = (A + B + C) / 2.0
      AREA = SQRT( S * (S - A) * (S - B) * (S - C))
      WRITE(6,601) A,B,C,AREA
  601 FORMAT(4H A= ,I5,5H  B= ,I5,5H  C= ,I5,8H  AREA= ,F10.2,12HSQUARE UNITS)
      STOP
      END
```



FORTRAN = FORMULA TRANSLATOR

Most features common to programming languages are introduced.

User subroutines are introduced in FORTRAN II (1958).

Interestingly, FORTRAN was sold as a high-level language from which efficient computer code would be generated. Nowadays we thing of high level programs as being “the code”.

ALGOL — 1958

```
begin
   ...
end
```

Block structure

Recursion

BNF

```
<statement> ::= <unconditional statement>
     | <conditional statement>
     | <for statement>
...
```

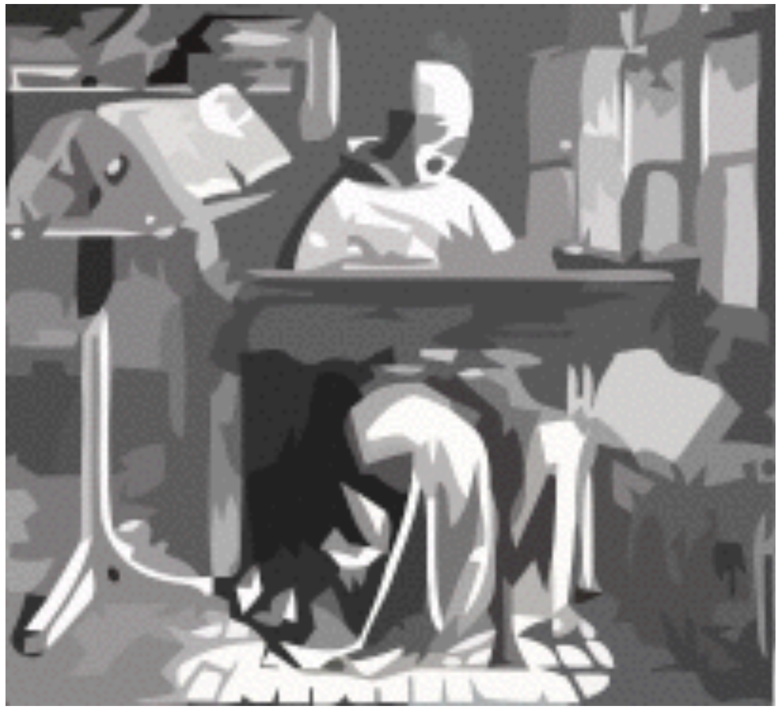
ALGOL introduced numerous important innovations, including:

- Backus-Naur form to formally specify the language grammar
- Recursion (FORTRAN had none)
- Block structure (as in *all* modern PLs)

Even though FORTRAN survives even today, and no one programs on ALGOL anymore, ALGOL has been far more influential, and its impact is recognizable in all PLs used today.

Lisp — 1958

```
(defun factorial (n)
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```



Programs as data

Garbage collection

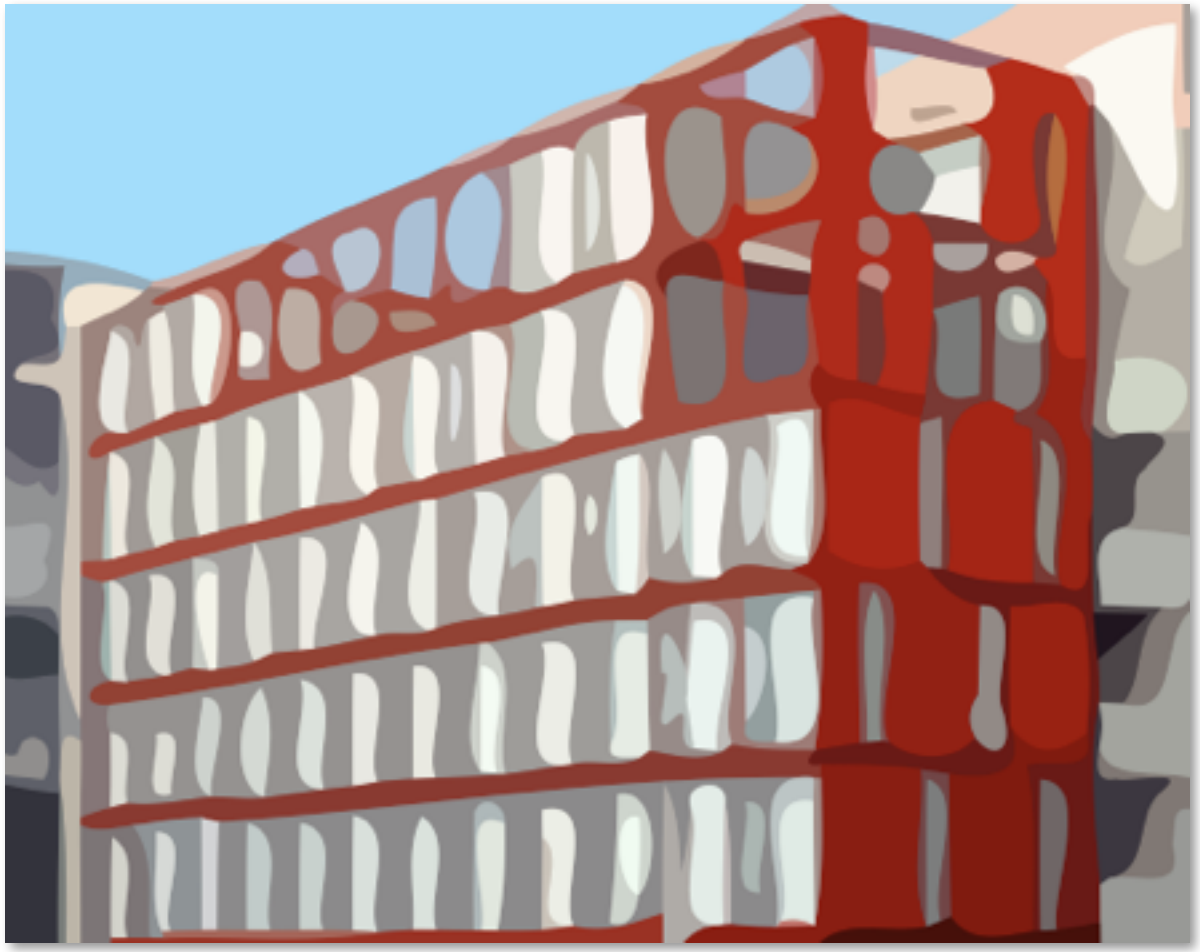
Lisp introduced the notion of *symbolic computation*: everything in Lisp is a list of data, some of which may also be lists. Interestingly, *Lisp programs are also represented as lists*.

This makes it easy to implement a Lisp interpreter in Lisp itself.

Lisp is the mother of all interactive and dynamic languages.

COBOL — 1959

```
ADD YEARS TO AGE.
MULTIPLY PRICE BY
QUANTITY GIVING COST.
SUBTRACT DISCOUNT FROM
COST GIVING FINAL-COST.
```



modules

A key motivation for Cobol was that it should be readable (for managers) but ended it up just being verbose.

Even today, a large portion of business software is implemented in Cobol. *[How much? Who knows?]*

Cobol’s main innovation was in supporting modular programming.

BASIC — 1964

```
10 INPUT "What is your name: ", U$
20 PRINT "Hello "; U$
30 INPUT "How many stars do you want: ", N
40 S$ = ""
50 FOR I = 1 TO N
60 S$ = S$ + "*"
70 NEXT I
80 PRINT S$
90 INPUT "Do you want more stars? ", A$
100 IF LEN(A$) = 0 THEN GOTO 90
110 A$ = LEFT$(A$, 1)
120 IF A$ = "Y" OR A$ = "y" THEN GOTO 30
130 PRINT "Goodbye "; U$
140 END
```

interactive programming for the masses

Basic was an interactive language for non-scientists.

Although it was invented in the 1960s, it became especially influential as a PL for hobbyists when the PC became popular in the 1980s.

JCL — 1964

```
//IS198CPY JOB (IS198T30500),'COPY JOB',CLASS=L,MSGCLASS=X
//COPY01   EXEC PGM=IEBGENER
//SYSPRINT DD SYSOUT=*
//SYSUT1   DD DSN=OLDFILE,DISP=SHR
//SYSUT2   DD DSN=NEWFILE,
//            DISP=(NEW,CATLG,DELETE),
//            SPACE=(CYL,(40,5),RLSE),
//            DCB=(LRECL=115,BLKSIZE=1150)
//SYSIN    DD DUMMY
```

invented *scripting* for IBM 360

A *scripting language* instructs one or more “actors” to perform some set of actions according to a “script.”

The language does not need to be computationally complete. Expressive power is achieved by the fact that the actors have very powerful actions they can perform.

JCL is arguably the first such scripting language, designed to script the execution of batch jobs on a mainframe computer.

JCL was horribly complex – this verbose script just copies a file.

Planner — 1969
Prolog — 1972

```
man(socrates).
mortal(X) :- man(X).
```

Facts and rules

```
?- mortal(socrates).
Yes
```

Queries and inferences

```
?- mortal(elvis).
No
```

The logic programming paradigm is based on the notion that knowledge can be expressed as a database of *facts*, and a set of *rules* for inferring new facts. Given a logic program, you can then pose a *query* to determine whether a given fact can be deduced.

The inference engine will then try to deduce the desired result (the "goal"), applying facts and rules, backtracking when it reaches a dead end, and either concluding with a happy result, or failure if the goal cannot be reached.

In this case, we cannot deduce that elvis is mortal, so we conclude with failure.

Logic programming is good for *search problems*, where a solution must be found in a large search space.

Pascal — 1970

Supports *structured* programming

```
function gcd (a, b: integer) : result real;
   var x : integer;
begin
   if b= 0 then gcd := a
   else
     begin
       x := a;
       while (x >= b) do
         begin
           x := x - b
         end;
         gcd := gcd(b,x)
     end
end
```

Successful with PCs

Pascal was designed by Niklaus Wirth at ETHZ as a teaching language, but it really became popular in the 1980s as the language for programming PCs. (TurboPascal was the cutting edge implementation.)

Pascal promoted clean, structured programming.

Pascal's type system was too restrictive, though, forcing the same code to be rewritten to handle similar but different types.

C — 1972

Bridging low- and high-level programming

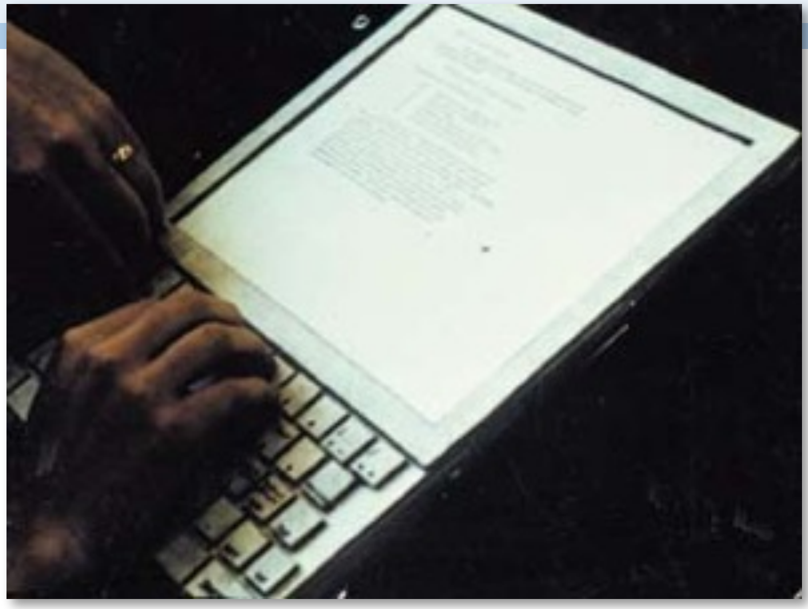
```
#include <stdio.h>
//echo the command line arguments
int main (int argc, char* argv[]) {
   int i;
   for (i=1; i<argc; i++) {
      printf("%s ", argv[i]);
   }
   printf("\n");
   return 0;
}
```

Good for portable systems programming

C is a high-level imperative 3GL designed to be close to machine code. It is heavily used as a “high-level assembler”.

Its main success has been in developing highly portable Unix and Linux code.

Smalltalk — 1972



"Dynabook" vision

Everything is an object

Everything happens by sending messages

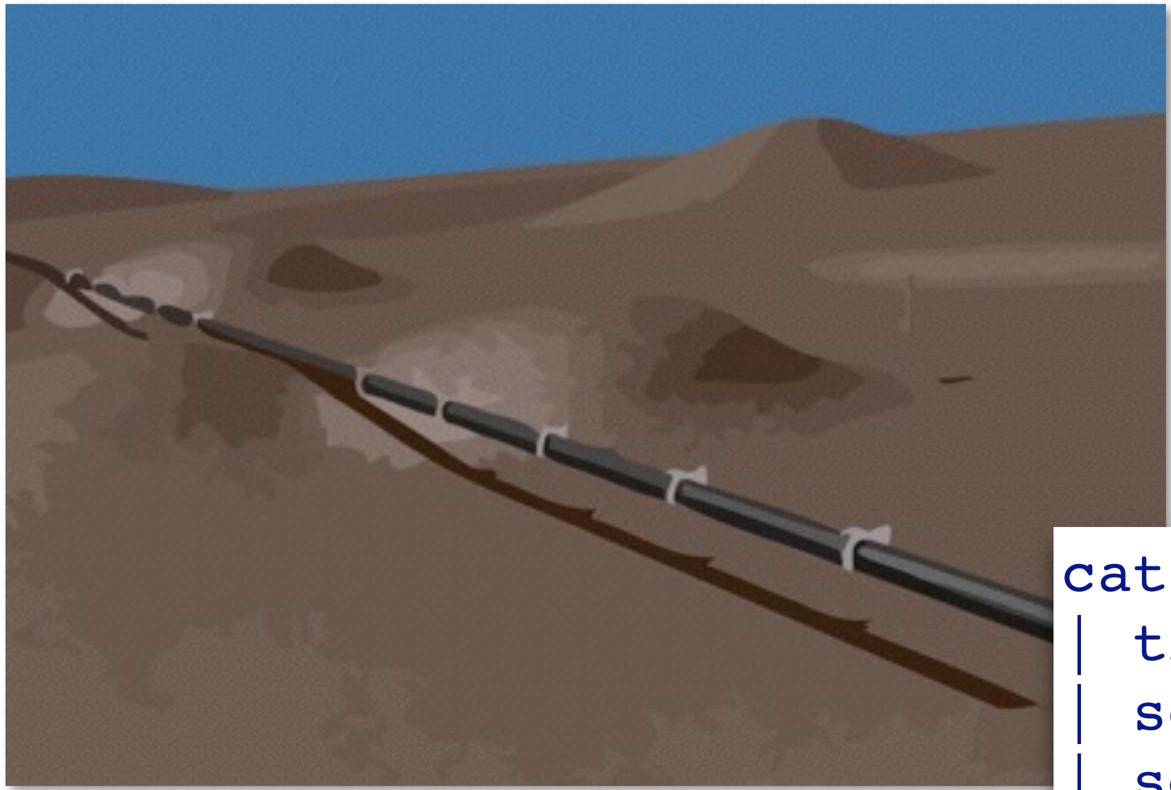
```
5 factorial → 120
```

```
Integer»factorial
    self = 0 ifTrue: [^ 1].
    self > 0 ifTrue: [^ self * (self - 1) factorial].
    self error: 'Not valid for negative integers'
```

Object-oriented programming was originally invented in the early 1960s as an add-on to procedural languages for simulating real world objects.

Smalltalk was the first language and *system* to use objects as the foundation for programming. (In Smalltalk, *everything* is an object.)

Bourne shell — 1977



Scripting *pipelines* of commands

```
cat Notes.txt
| tr -c '[:alpha:]' '\012'
| sed '/^$/d'
| sort
| uniq –c
| sort –rn
| head -5
```

```
14 programming
14 languages
 9 of
 7 for
 5 the
```

The Bourne Shell was designed as the original shell & scripting language for Unix. You cannot really program in the shell; you can only glue together Unix commands, each of which performs a dedicated task.

The Unix commands themselves are typically programmed in C.

Interestingly, shell pipelines are concurrent programs, since each filter in the chain runs as a separate Unix process. The O/S handles all the concurrency control by managing the flow of data between the processes.

SQL — 1978

Domain-specific language for relational databases

```
SELECT *
   FROM Book
   WHERE price > 100.00
   ORDER BY title;
```

SQL is not just a query language – it is also used for manipulating and updating tables.

However it is not a full-blown programming language. It is designed specifically for the domain of querying and manipulating relational database tables.

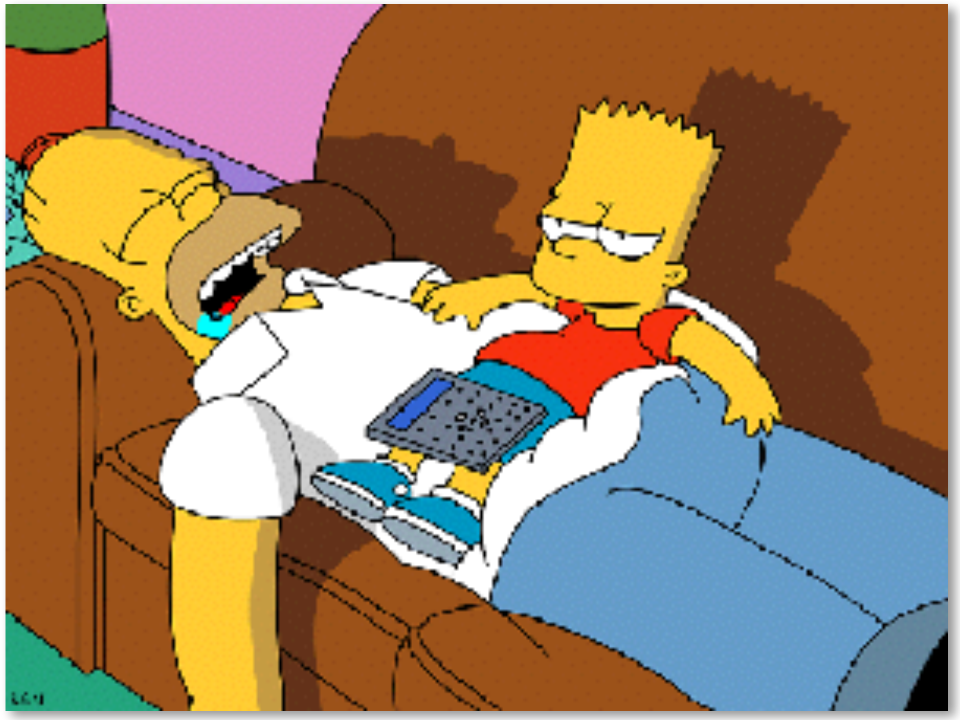
Miranda — 1985

“Pure” functional programming

```
fibs = 1 : 1 : fibsAfter 1 1
fibsAfter a b = (a+b) : fibsAfter b (a+b)
```

```
take 10 fibs
→ [1,1,2,3,5,8,13,21,34,55]
```

Lazy evaluation



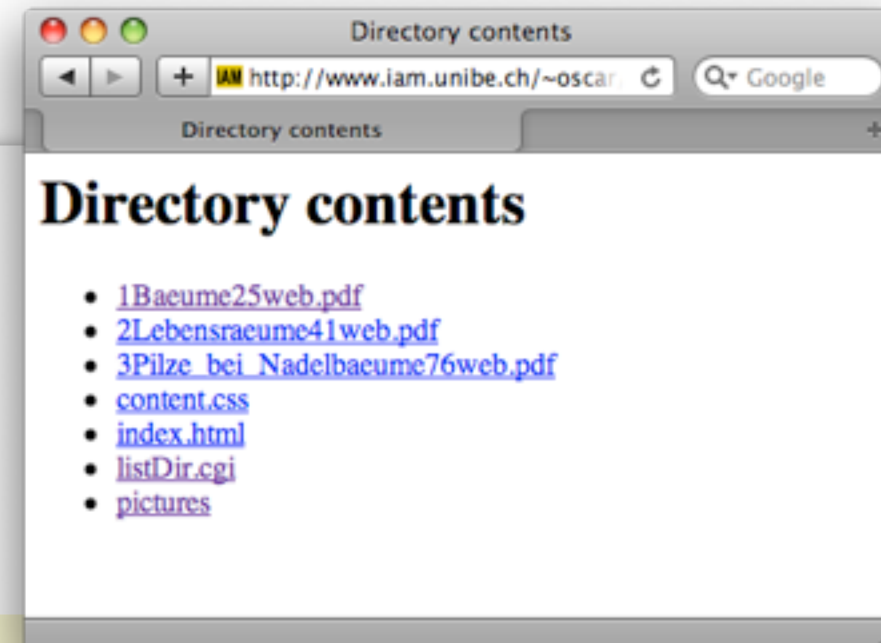
This example is written in Haskell, a successor to Miranda.

`fibs` is an infinite, lazy list. Values are only computed if and when needed. Lazy evaluation has been highly influential in the design of modern functional languages.

The technique is also used to great effect in mainstream languages. For example, a complex user interface can be lazily loaded to give the illusion of speed and responsiveness to the end user (only actually load parts when you actually need them).

Perl — 1987
CGI — 1993

```
#!/usr/bin/perl -w
print "Content-type: text/html\n\n";
print <<'eof'
<html><head><title>Directory contents</title></head>
<body>
<h1>Directory contents</h1><ul>
eof
;
@files = <*>;
foreach $file (@files) {
    print '<li><a href="' . $file . '">' . $file . "</li>\n";
}
print "</ul></body></html>\n";
__END__
```



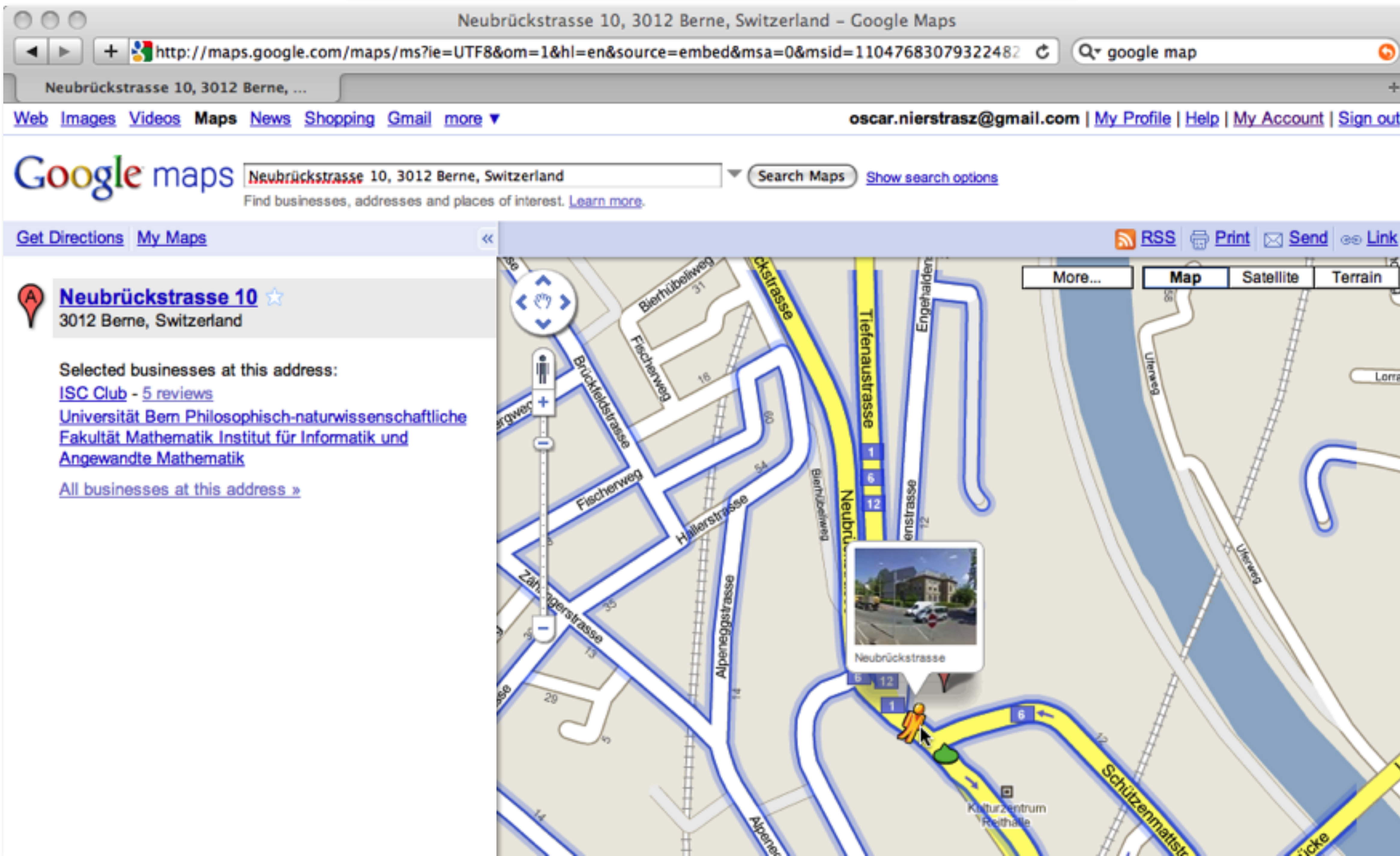
Text manipulation, then server-side *web scripting*

Perl is a text-manipulation language that doubles as a scripting language. It supports many dedicated and efficient features for reading, rewriting, and outputting text.

Perl really came into its own after 1993 when it started being used for *server-side scripts* (CGI scripts).

JavaScript — 1995
AJAX — 2005

Client-side browser scripting



JavaScript on the other hand introduced *client-side scripting*. A web page containing JavaScript code code can request the browser to perform various actions.

It was not until relatively recently (ca. 2005) that its potential was realized to make web pages truly interactive (e.g., Google maps), with the help of Ajax and related technologies.

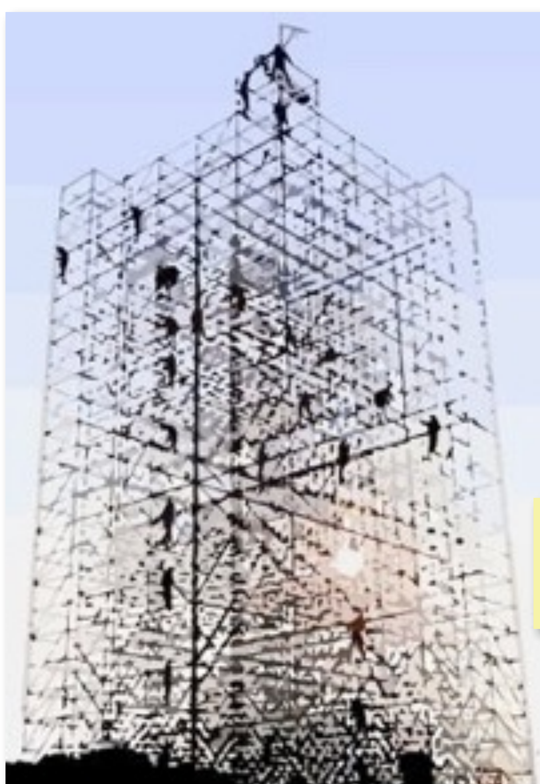
Roadmap

- What is a programming language?
- Historical Highlights
- **Conclusions**

How do these languages differ?

Imperative

Functional



Logic

Object-oriented

These paradigms can be summarized as follows:

— Imperative: data + algorithms

— Functional: stateless; functions pass values to each other

— OOP: objects send messages to each other

— Logic: facts + rules → new facts

Conclusions

Programming is *modeling*

Programming languages have always evolved to bring programming *closer to the users' problems*

We are still *very early* in the history of programming