Principles of Programming Languages
Lecture 1: Introduction

Andrei Arusoaie[1]

[1]Department of Computer Science

Fall 2021

Outline

Principles of Programming Languages - Motivation

- Learning a new PL is nowadays a requirement
- “Programmers need to move from one PL to another with naturalness and speed” [Gabbrielli2010]
- PLs have their similarities, analogies, inherited characteristics

GOAL:

understand the basic mechanisms behind the creation of a PL

Principles of Programming Languages - Motivation

- Learning a new PL is nowadays a requirement
- “Programmers need to move from one PL to another with naturalness and speed” [Gabbrielli2010]
- PLs have their similarities, analogies, inherited characteristics

GOAL:

understand the basic mechanisms behind the creation of a PL

Principles of Programming Languages - Motivation

- Learning a new PL is nowadays a requirement
- “Programmers need to move from one PL to another with naturalness and speed” [Gabbrielli2010]
- PLs have their similarities, analogies, inherited characteristics

GOAL:

understand the basic mechanisms behind the creation of a PL

Principles of Programming Languages - Motivation

- Learning a new PL is nowadays a requirement
- “Programmers need to move from one PL to another with naturalness and speed” [Gabbrielli2010]
- PLs have their similarities, analogies, inherited characteristics

GOAL:

understand the basic mechanisms behind the creation of a PL

Skills to develop

- The definition of a programming language:

  Syntax + *Semantics*

- Have a better understanding of the PLs you already know
- Ability to learn a new PL fast
- Get familiar with a framework for defining languages

Skills to develop

- The definition of a programming language:

  Syntax + *Semantics*

- Have a better understanding of the PLs you already know
- Ability to learn a new PL fast
- Get familiar with a framework for defining languages

Skills to develop

- The definition of a programming language:

$$\textit{Syntax} + \textit{Semantics}$$

- Have a better understanding of the PLs you already know
- Ability to learn a new PL fast
- Get familiar with a framework for defining languages

Skills to develop

- The definition of a programming language:

$$\textit{Syntax} + \textit{Semantics}$$

- Have a better understanding of the PLs you already know
- Ability to learn a new PL fast
- Get familiar with a framework for defining languages

Principles of Programming Languages - Organisation

- Team: Andrei Arusoaie
- Period: 1^{st} semester (Fall 2021), 2^{nd} year
- https://www.uaic.ro/studii/structura-anului-universitar/
- 14 (full activity) weeks
- Final grade = 20 (lab assignments) + 80 (2 exams)
- Assignments: each lab has a dedicated assignment that needs to be finished at the end of the lab
- Exams: 80 pts = 40 pts(week #8) + 40 pts(week #14)
- Bonuses: interesting solutions

Principles of Programming Languages - Organisation

- Team: Andrei Arusoaie
- Period: 1^{st} semester (Fall 2021), 2^{nd} year
- `https://www.uaic.ro/studii/structura-anului-universitar/`
- 14 (full activity) weeks
- Final grade = 20 (lab assignments) + 80 (2 exams)
- Assignments: each lab has a dedicated assignment that needs to be finished at the end of the lab
- Exams: 80 pts = 40 pts(week #8) + 40 pts(week #14)
- Bonuses: interesting solutions

Principles of Programming Languages - Organisation

- Team: Andrei Arusoaie
- Period: 1^{st} semester (Fall 2021), 2^{nd} year
- `https://www.uaic.ro/studii/structura-anului-universitar/`
- 14 (full activity) weeks
- Final grade = 20 (lab assignments) + 80 (2 exams)
- Assignments: each lab has a dedicated assignment that needs to be finished at the end of the lab
- Exams: 80 pts = 40 pts(week #8) + 40 pts(week #14)
- Bonuses: interesting solutions

Principles of Programming Languages

Course description:

- RO: https://profs.info.uaic.ro/~arusoaie.andrei/lectures/PLP/2021/Fisa_disciplinei_PLP.pdf
- EN: https://profs.info.uaic.ro/~arusoaie.andrei/lectures/PLP/2021/Fisa_disciplinei_PLP_en.pdf

Fall 2021:

- https://profs.info.uaic.ro/~arusoaie.andrei/lectures/PLP/2021/plp.html

Bibliography

1. Software Foundations - Volume 2, Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, Andrew Tolmach, Brent Yorgey
   https://softwarefoundations.cis.upenn.edu/plf-current/index.html
2. Programming Languages: Principles and Paradigms, Maurizio Gabbrielli, Simone Martini
   http://websrv.dthu.edu.vn/attachments/newsevents/content2415/Programming_Languages_-_Principles_and_Paradigms_thereds1106.pdf
3. Practical Foundations of Programming Languages, Robert Harper
   https://www.cs.cmu.edu/~rwh/pfpl/2nded.pdf
4. The formal semantics of Programming Languages – An Introduction, Glynn Winskell
   https://www.cin.ufpe.br/~if721/intranet/TheFormalSemanticsofProgrammingLanguages.pdf

Labs

- Lab: Coq
- Web: `https://coq.inria.fr/`
- Software Foundations - Volume 2, Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, Andrew Tolmach, Brent Yorgey

  `https://softwarefoundations.cis.upenn.edu/plf-current/index.html`

Labs

- Lab: Coq
- Web: `https://coq.inria.fr/`
- Software Foundations - Volume 2, Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, Andrew Tolmach, Brent Yorgey

  `https://softwarefoundations.cis.upenn.edu/plf-current/index.html`

Today's lecture

1. History
2. Programming paradigms
3. What's next?

Today's lecture

1. History
2. Programming paradigms
3. What's next?

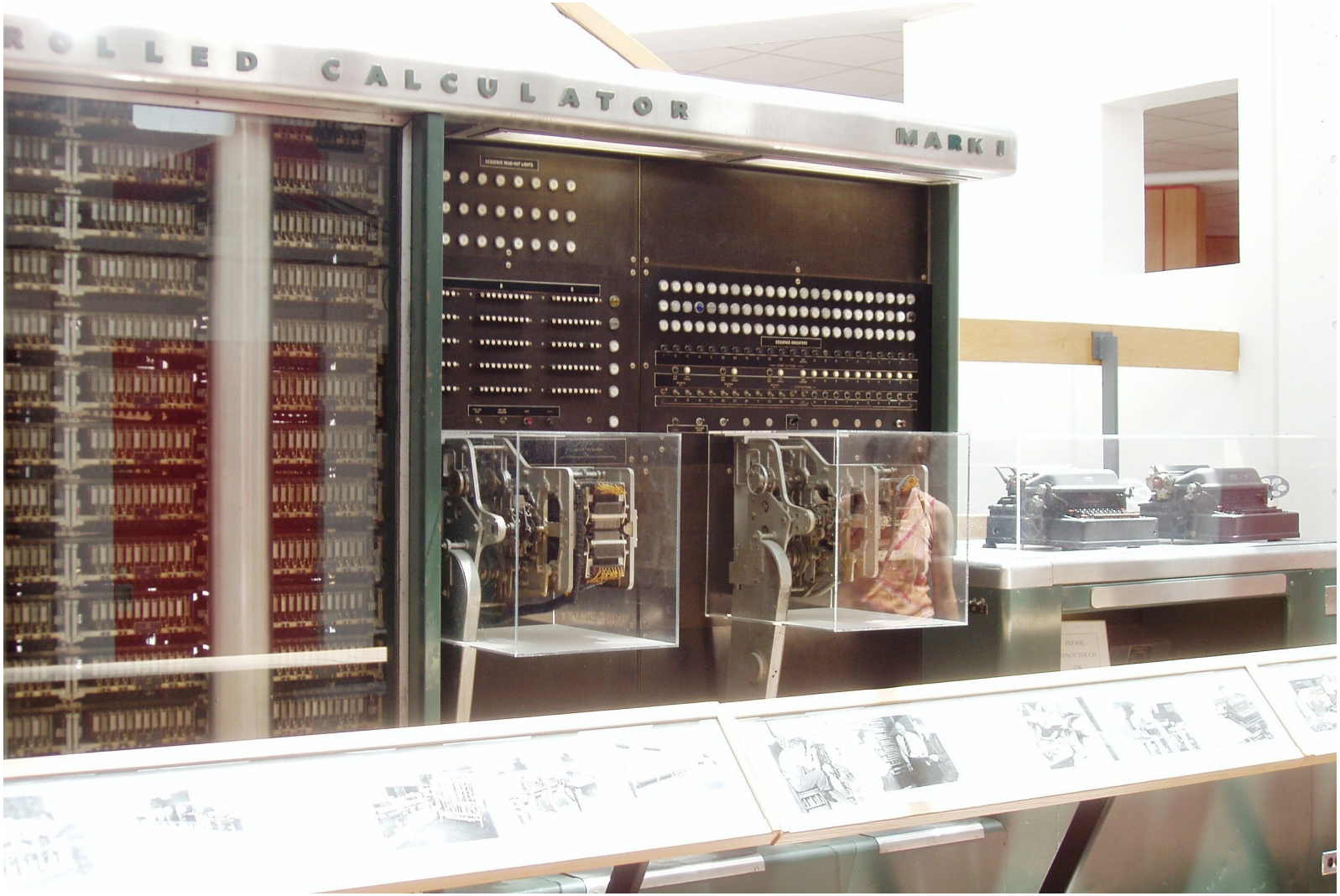
Today's lecture

1. History
2. Programming paradigms
3. What's next?

ASCC/MARKI

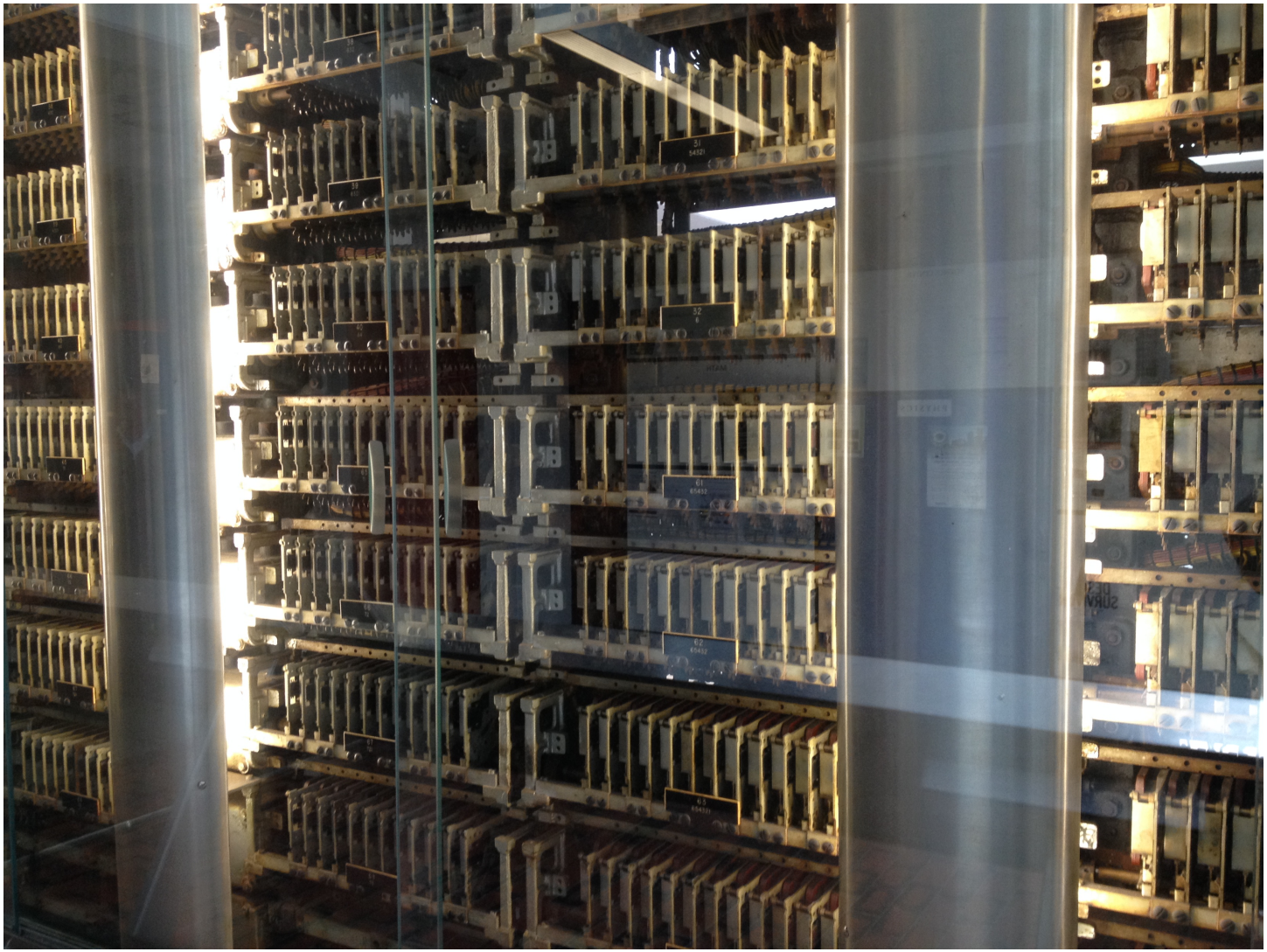
- Year: 1944
  - Automatic Sequence Controlled Calculator
  - built by IBM and Harward Univ., H. Aiken
  - programmed using punched tapes and external physical media (like switches)
  - 10 m length, 4 tones, very expensive

ASCC/MARK I: The computer



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

ASCC/MARK I: Computing section

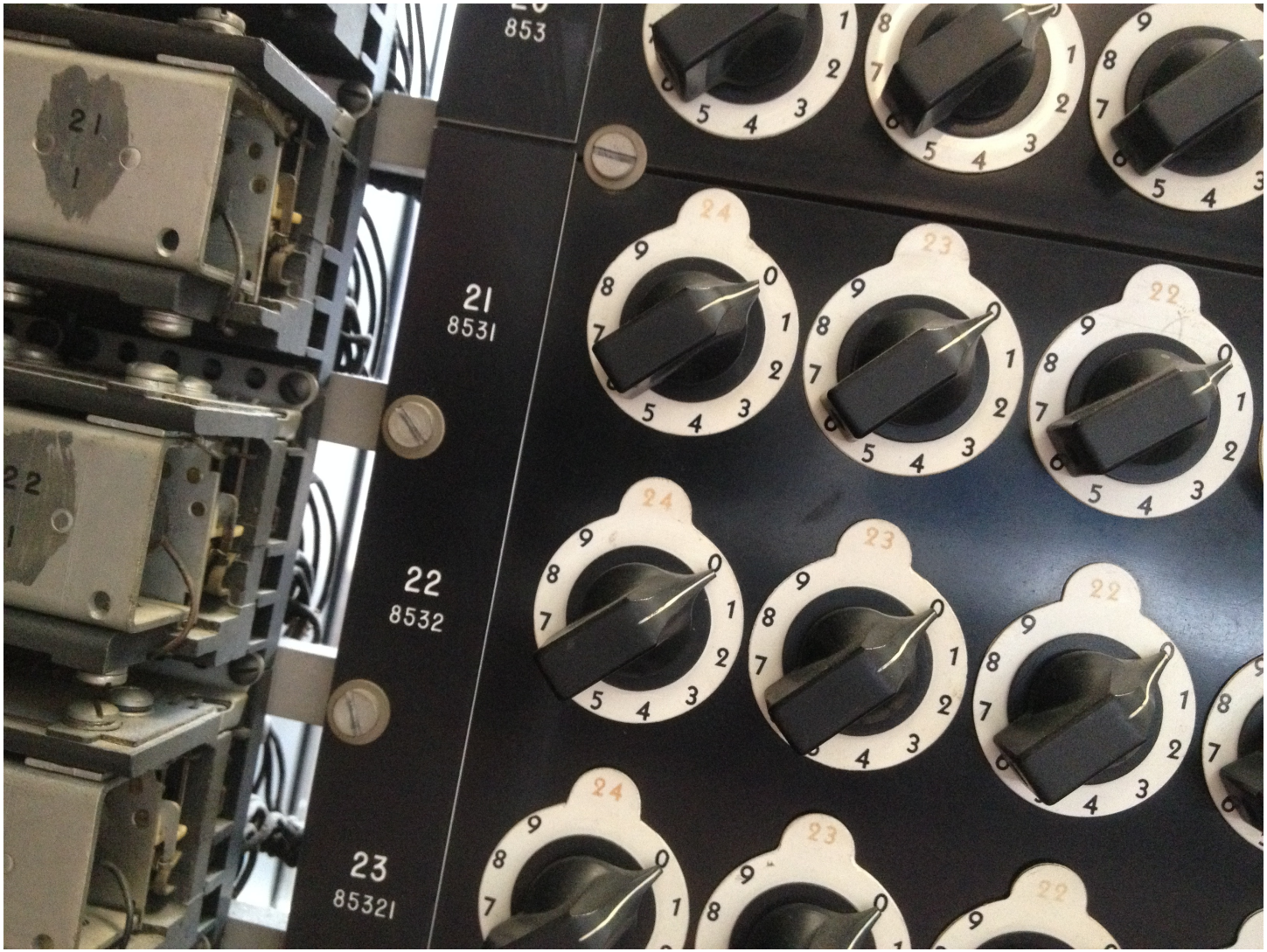


Source: `https://en.wikipedia.org/wiki/Harvard_Mark_I`

ASCC/MARK I: Sequence indicators

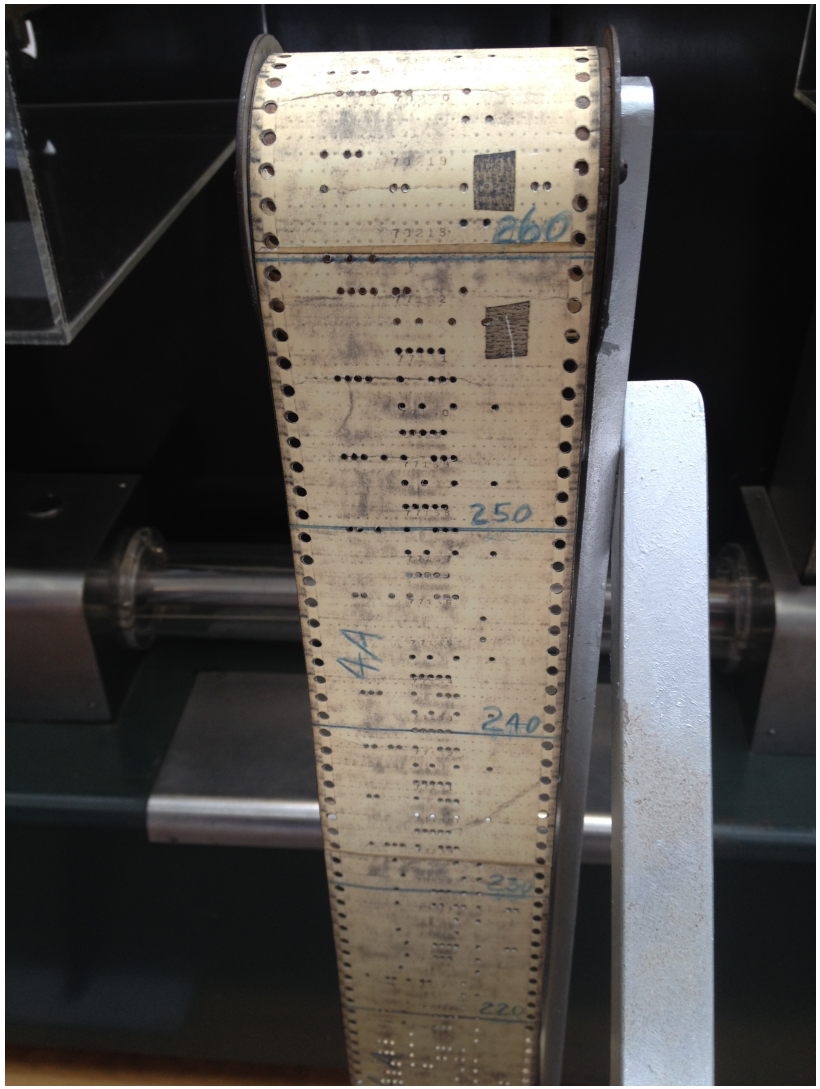
Source: `https://en.wikipedia.org/wiki/Harvard_Mark_I`

ASCC/MARK I: Switches for program data constants



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

ASCC/MARK I: Punched tape



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

ASCC/MARK I: Punched tape

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

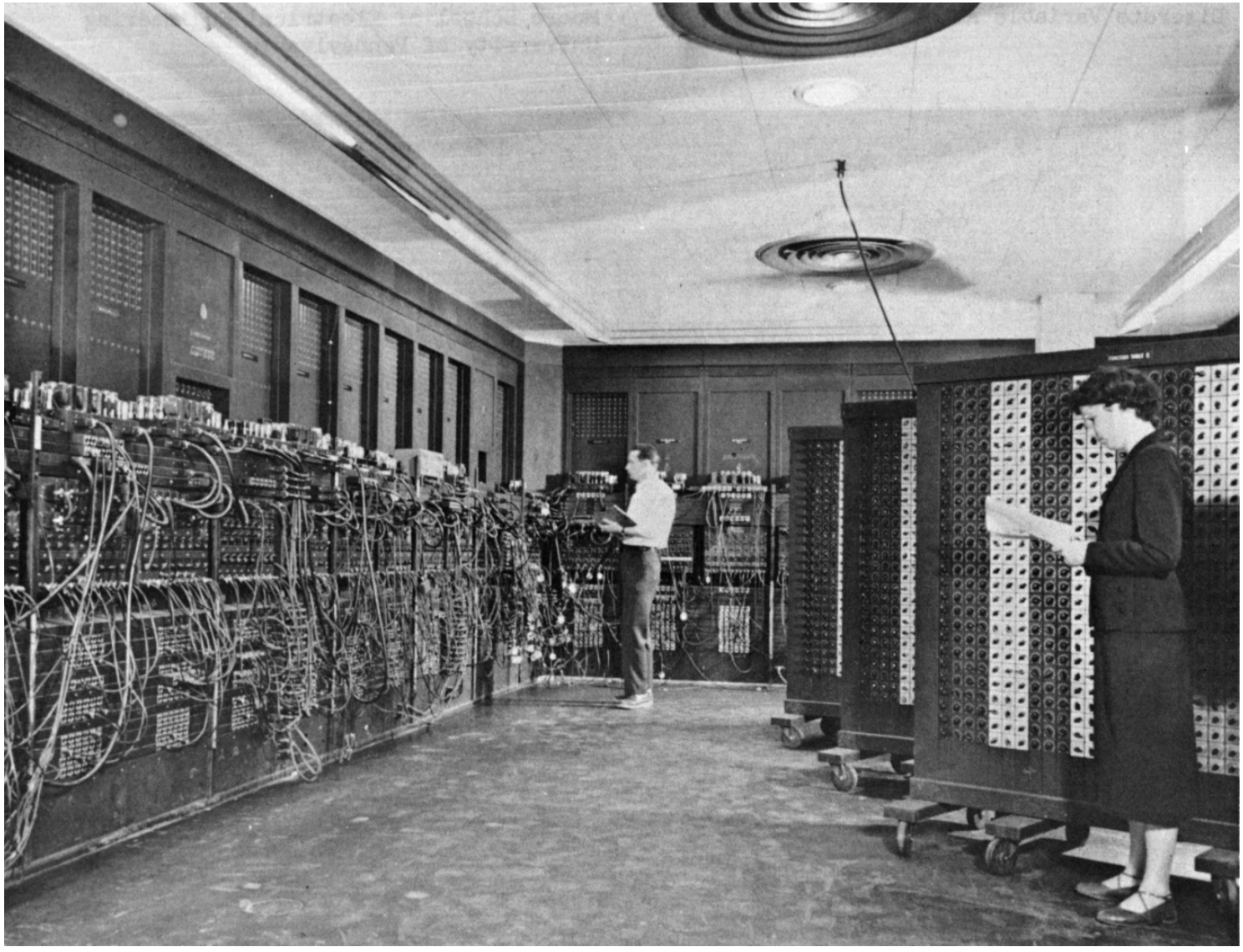
ASCC/MARK I - Interesting facts

- 60 sets of 24 swtiches for manual data entry
- 72 numbers storage, 23 decimal digits long
- 3 additions/subtractions per second
- 1 multiplication took 6 seconds
- 1 division took 15.3 seconds
- over a minute: logarithmic or trigonometry function
- instructions: 24-channel punched tape
- input numbers: stored on a different tape with a different format
- loops: join the end of the tape back to the beginning of the tape
- ... implosion of the atomic bomb

ENIAC

- Year: 1946
  - Electronic Numerical Integrator and Computer
  - J. Mauchly and J.P. Eckert, early (design) J. von Neumann
  - Orders of magnitude faster than humans at computing
  - no storage
  - programmed using external physical media (electrical cables)
  - used to calculate artillery firing tables for US Army, ballistic trajectories

ENIAC



Source: https://en.wikipedia.org/wiki/ENIAC

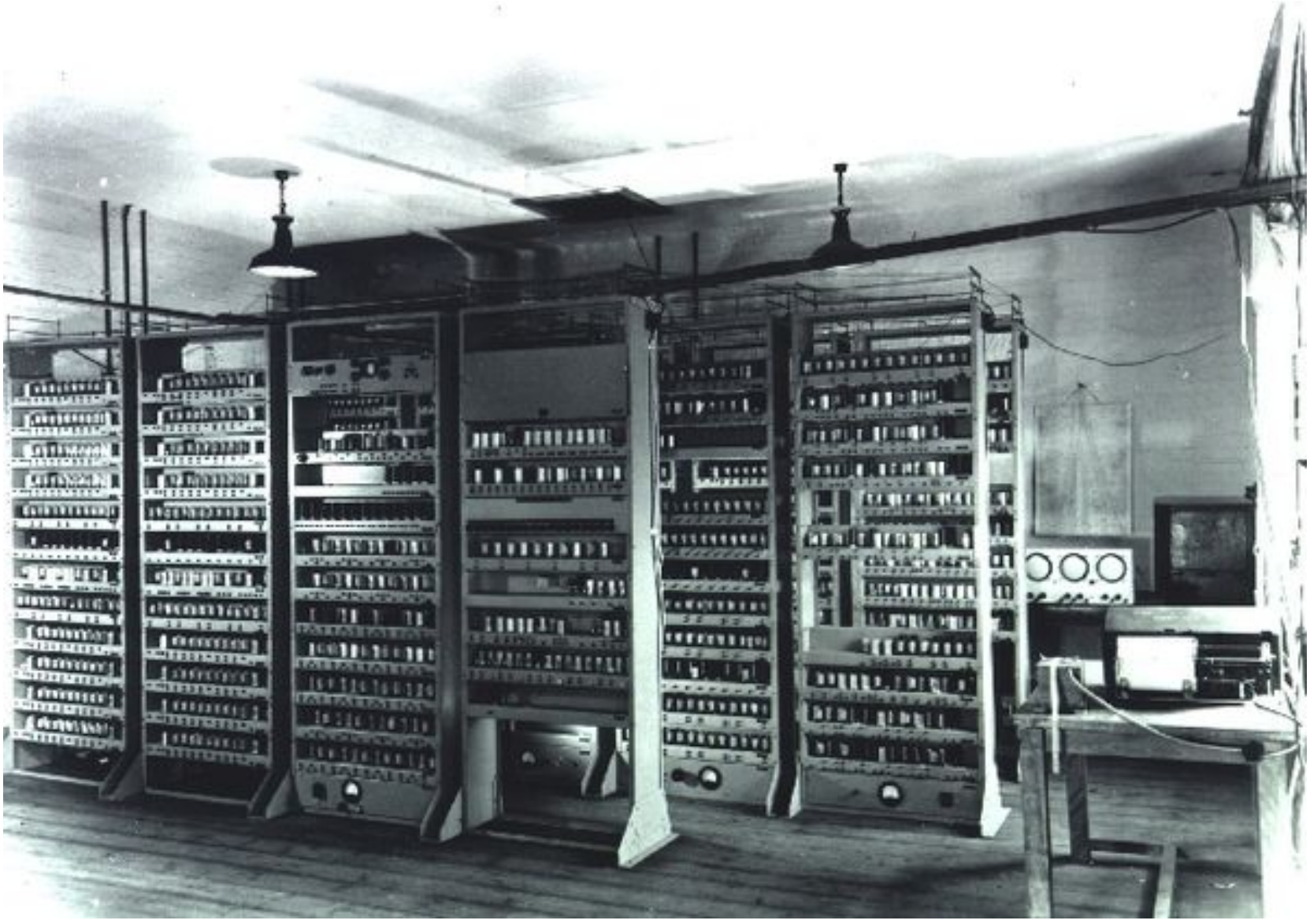
ENIAC

- ENIAC was able to compute in 30 sec a trajectory that took a human 20 hours to compute
- Price: $500000 (the equivalent of $7.3 m in 2020)
- The computer was made of individual panels, each panel performed different functions
- 18000 vaccum tubes, 7200 crystal diodes, 1500 relays, 70000 rezistors, 10000 capacitors
- Weight 30 tones, Occupied surface: 167 m^2
- High energy consumption
- input: IBM card readers, later: magentic core memory
- Arithmetic: count pulses on ring counters; 5k add/sub per second for 10 digit numbers
- Division or square root: 35 per second
- Loops, branching, subroutines
- Hidrogen bomb...

EDSAC

- Electronic Delay Storage Automatic Calculator
- Year: 1949
- The first “computer” in the modern sense
  - electronic and digital
  - able to perform four elementary arithmetic operations
  - programable
  - storage of programs and data

EDSAC



Source: `https://en.wikipedia.org/wiki/EDSAC`

EDSAC

- Input: five-hole punched tape
- Cycle time: 1.5 ms for ordinary instructions + 6 ms for multiplication
- Output: teleprinter
- Memory: 512 x 18-bit words, later 1024 words -> 800 words programs
- Instructions: add, subtract, multiply, shifts, store, goto, read, prin, round, stop, no-op

Generations

- 1GL (first generation languages): machine language
  - elementary instructions immediately executable by a processor
  - coding: completely manual
  - hard to use, too “machine connected”
  - people realized that they need something closer to the user’s natural language
- 2GL: (second generation languages): assembly language
  - introduced to ease the development
  - symbolic representation of the machine language
  - translated into machine code
  - downside: each machine has its own assembly language

Generations

- 1GL (first generation languages): machine language
  - elementary instructions immediately executable by a processor
  - coding: completely manual
  - hard to use, too “machine connected”
  - people realized that they need something closer to the user’s natural language
- 2GL: (second generation languages): assembly language
  - introduced to ease the development
  - symbolic representation of the machine language
  - translated into machine code
  - downside: each machine has its own assembly language

Generations

- 1950 – 3GL: high-level languages
  - abstract languages
  - ignore the physical characteristics of the computer
  - well suited to express *algorithms*
- 1950: ALGOL = ALGOrithmic Languages (family of languages)
- 1957: FORTRAN = FORmula TRANslation
- 1960: LISP = LISt Processor
- However, the development was still an issue
- Lots of human resources required, expensive hardware

Generations

- 1950 – 3GL: high-level languages
  - abstract languages
  - ignore the physical characteristics of the computer
  - well suited to express *algorithms*
- 1950: ALGOL = ALGOrithmic Languages (family of languages)
- 1957: FORTRAN = FORmula TRANslation
- 1960: LISP = LISt Processor
- However, the development was still an issue
- Lots of human resources required, expensive hardware

Generations

- 1950 – 3GL: high-level languages
  - abstract languages
  - ignore the physical characteristics of the computer
  - well suited to express *algorithms*
- 1950: ALGOL = ALGOrithmic Languages (family of languages)
- 1957: FORTRAN = FORmula TRANslation
- 1960: LISP = LISt Processor
- However, the development was still an issue
- Lots of human resources required, expensive hardware

Generations

- 1970's: microprocessor; batch processing
- Further abstractions pushed PL development to what PL are today
- New paradigms: object-oriented, declarative programming
- The C language: Dennis Ritchie and Ken Thompson
  - Initially designed for the UNIX operating system
  - Successor of B
  - Followed by: Pascal, SmallTalk
- Declarative languages: ML (Meta Language), PROLOG

Generations

- 1970's: microprocessor; batch processing
- Further abstractions pushed PL development to what PL are today
- New paradigms: object-oriented, declarative programming
- The C language: Dennis Ritchie and Ken Thompson
  - Initially designed for the UNIX operating system
  - Successor of B
  - Followed by: Pascal, SmallTalk
- Declarative languages: ML (Meta Language), PROLOG

Generations

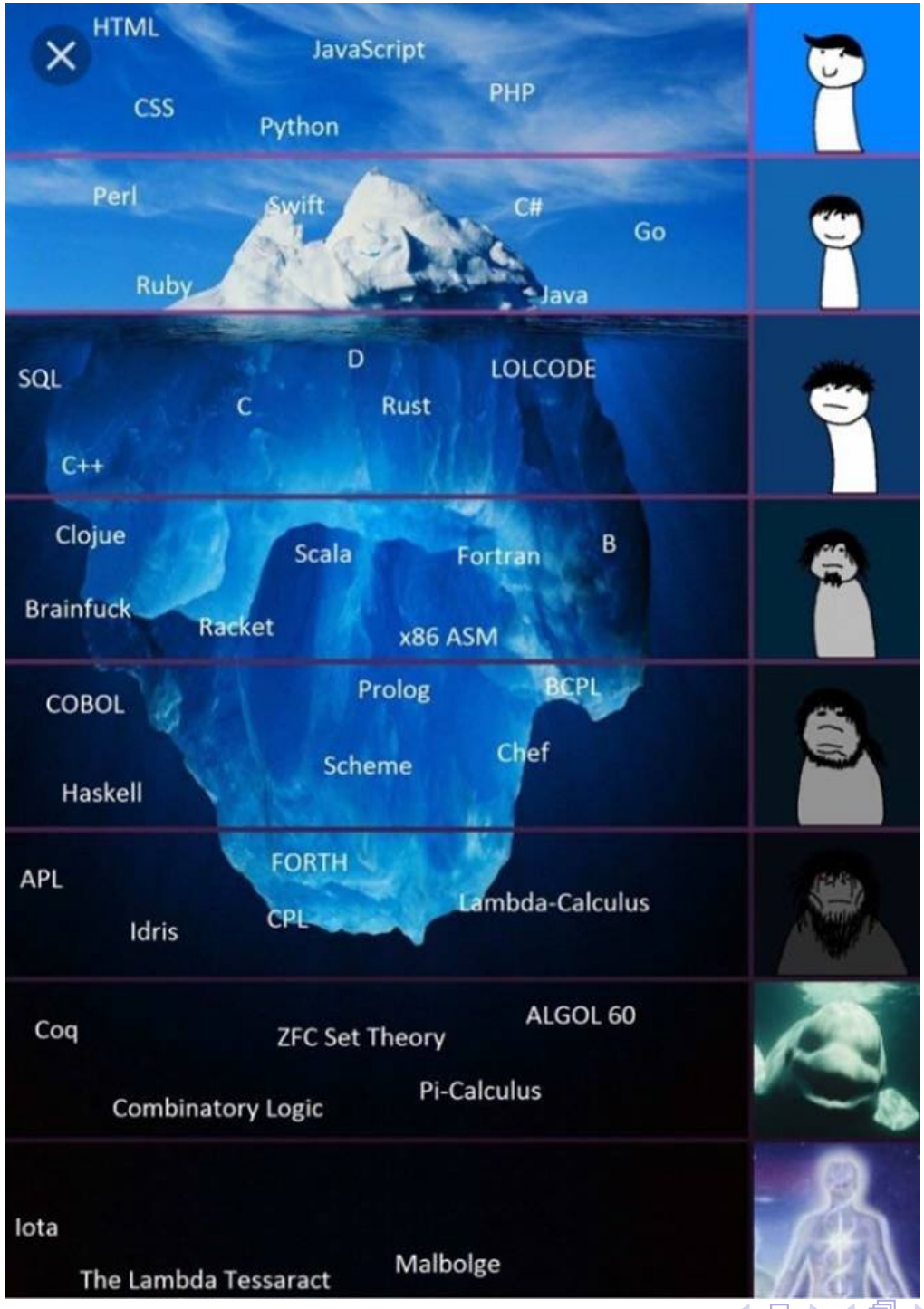
- 1970's: microprocessor; batch processing
- Further abstractions pushed PL development to what PL are today
- New paradigms: object-oriented, declarative programming
- The C language: Dennis Ritchie and Ken Thompson
  - Initially designed for the UNIX operating system
  - Successor of B
  - Followed by: Pascal, SmallTalk
- Declarative languages: ML (Meta Language), PROLOG

Generations

- 1980's: the PC era
- 1986: C++ - Bjarne Stroustrup
- 1989: HTML :(- T. Berners-Lee, Python - Guido van Rossum
- 1990's: internet, WWW
- 1990-1995: Java - Jim Gosling at SUN
- 1994 : PHP - Rasmus Lerdorf
- 2000 : (Microsoft) C# - Anders Hejlsberg
- 2004 : Scala - Martin Odersky
- 2012 : Go - at Google Inc. (started in 2007 by R. Griesemer, R. Pike, K. Thompson)
- 2014 : Swift - at Apple Inc. (started in 2010 by Chris Lattner)

Generations

- 1980's: the PC era
- 1986: C++ - Bjarne Stroustrup
- 1989: HTML :(- T. Berners-Lee, Python - Guido van Rossum
- 1990's: internet, WWW
- 1990-1995: Java - Jim Gosling at SUN
- 1994 : PHP - Rasmus Lerdorf
- 2000 : (Microsoft) C# - Anders Hejlsberg
- 2004 : Scala - Martin Odersky
- 2012 : Go - at Google Inc. (started in 2007 by R. Griesemer, R. Pike, K. Thompson)
- 2014 : Swift - at Apple Inc. (started in 2010 by Chris Lattner)


HTML
JavaScript
CSS
PHP
Python
Perl
Swift
C#
Go
Ruby
Java
SQL
D
LOLCODE
C
Rust
C++
Clojue
Scala
Fortran
B
Brainfuck
Racket
x86 ASM
COBOL
Prolog
BCPL
Chef
Scheme
Haskell
FORTH
APL
Lambda-Calculus
CPL
Idris
Coq
ZFC Set Theory
ALGOL 60
Pi-Calculus
Combinatory Logic
Iota
Malbolge
The Lambda Tessaract

Main programming paradigms

1. Imperative programming
   - *First do this and then do that*
2. Object-oriented programming
   - *Model the world using objects that exchange messages*
3. Functional programming[1]
   - *Evaluate an expression and pass the result*
4. Logic programming[1]
   - *Answer questions by searching a solution*

[1] Part of the Declarative programming paradigm

Main programming paradigms

1. Imperative programming
   - *First do this and then do that*
2. Object-oriented programming
   - *Model the world using objects that exchange messages*
3. Functional programming[1]
   - *Evaluate an expression and pass the result*
4. Logic programming[1]
   - *Answer questions by searching a solution*

[1] Part of the Declarative programming paradigm

Main programming paradigms

1. Imperative programming
   - *First do this and then do that*
2. Object-oriented programming
   - *Model the world using objects that exchange messages*
3. Functional programming[1]
   - *Evaluate an expression and pass the result*
4. Logic programming[1]
   - *Answer questions by searching a solution*

[1] Part of the Declarative programming paradigm

Main programming paradigms

1. Imperative programming
   - *First do this and then do that*
2. Object-oriented programming
   - *Model the world using objects that exchange messages*
3. Functional programming[1]
   - *Evaluate an expression and pass the result*
4. Logic programming[1]
   - *Answer questions by searching a solution*

---

[1] Part of the Declarative programming paradigm

Imperative programming

Example program:

```
read(n);
s = 0;
if (n < 0) {
   print "error";
} else {
  while (n > 0) do {
    s = s + n;
    n = n - 1;
  }
}
```

- Requires a *program state*
- Execution is (incrementally) changing the program state
- Instructions: assignment, decisional, loops
- *First class value*: variable

Imperative programming

Example program:

```
read(n);
s = 0;
if (n < 0) {
   print "error";
} else {
  while (n > 0) do {
    s = s + n;
    n = n - 1;
  }
}
```

- Requires a *program state*
- Execution is (incrementally) changing the program state
- Instructions: assignment, decisional, loops
- *First class value*: variable

Imperative programming

Example program:

```
read(n);
s = 0;
if (n < 0) {
   print "error";
} else {
  while (n > 0) do {
    s = s + n;
    n = n - 1;
  }
}
```

- Requires a *program state*
- Execution is (incrementally) changing the program state
- Instructions: assignment, decisional, loops
- *First class value*: variable

Imperative programming

Example program:

```
read(n);
s = 0;
if (n < 0) {
   print "error";
} else {
  while (n > 0) do {
    s = s + n;
    n = n - 1;
  }
}
```

- Requires a *program state*
- Execution is (incrementally) changing the program state
- Instructions: assignment, decisional, loops
- *First class value*: variable

Imperative programming

Example program:

```
read(n);
s = 0;
if (n < 0) {
   print "error";
} else {
  while (n > 0) do {
    s = s + n;
    n = n - 1;
  }
}
```

- Requires a *program state*
- Execution is (incrementally) changing the program state
- Instructions: assignment, decisional, loops
- *First class value*: variable

Object-Oriented Programming

Example (Java):

```
public class Person {
  private String name;
  Person(String name) {
    this.name = name;
  }
  public String getName() {
    return name;
  }
}

// in the main function
  Driver john = new Driver("John");
  john.startDriving();
  boolean status = john.getStatus();
```

```
public class Driver extends Person
{
  private boolean isDriving;
  Driver(String name) {
    super(name);
    isDriving = false;
  }
  public void startDriving() {
    isDriving = true;
  }
  public boolean getStatus() {
    return isDriving;
  }
}
```

Object-Oriented Programming

- Model using *objects* and *classes*
- Objects exchange messages (dispatch/message passing)
- Abstraction, Encapsulation
- Inheritance, Polymorphism, Overriding, Overloading

Functional programming

Example:
(Haskell)

```
mysum :: Integer -> Integer
mysum n = if (n < 0)
          then 0
          else n + mysum (n - 1)
```

Run:

```
> mysum 10
55
> mysum 0
0
> mysum (-3)
0
```

- Computations = evaluation of mathematical functions
- No state!
- *Non-mutable* values
- First class value: function

Functional programming

Example:
(Haskell)

```
mysum :: Integer -> Integer
mysum n = if (n < 0)
          then 0
          else n + mysum (n - 1)
```

Run:

```
> mysum 10
55
> mysum 0
0
> mysum (-3)
0
```

- Computations = evaluation of mathematical functions
- No state!
- *Non-mutable* values
- First class value: function

Functional programming

Example:
(Haskell)

```
mysum :: Integer -> Integer
mysum n = if (n < 0)
          then 0
          else n + mysum (n - 1)
```

Run:

```
> mysum 10
55
> mysum 0
0
> mysum (-3)
0
```

- Computations = evaluation of mathematical functions
- No state!
- *Non-mutable* values
- First class value: function

Logic Programming

Example (Prolog):

```
man(john).
man(dan).
man(mark).
father(john, mark).
father(mark, dan).

grandfather(X,Y) :- man(X),
 man(Y),father(X,Z),father(Z,Y).
```

Search solution:

```
1 ?- father(mark, dan).
true.
2 ?- grandfather(mark, dan).
false.
3 ?- grandfather(john, dan).
true.
4 ?- grandfather(X, dan).
X = john .
```

- Algorithm = Logic + Control
- Programmer provides a logical specification
- An interpreter searches for the solution using resolution

Logic Programming

Example (Prolog):

```
man(john).
man(dan).
man(mark).
father(john, mark).
father(mark, dan).

grandfather(X,Y) :- man(X),
 man(Y),father(X,Z),father(Z,Y).
```

Search solution:

```
1 ?- father(mark, dan).
true.
2 ?- grandfather(mark, dan).
false.
3 ?- grandfather(john, dan).
true.
4 ?- grandfather(X, dan).
X = john .
```

- Algorithm = Logic + Control
- Programmer provides a logical specification
- An interpreter searches for the solution using resolution

Logic Programming

Example (Prolog):

```
man(john).
man(dan).
man(mark).
father(john, mark).
father(mark, dan).

grandfather(X,Y) :- man(X),
 man(Y),father(X,Z),father(Z,Y).
```

Search solution:

```
1 ?- father(mark, dan).
true.
2 ?- grandfather(mark, dan).
false.
3 ?- grandfather(john, dan).
true.
4 ?- grandfather(X, dan).
X = john .
```

- Algorithm = Logic + Control
- Programmer provides a logical specification
- An interpreter searches for the solution using resolution

Domain Specific Languages

- Web: (markup) HTML, CSS, scripting (e.g., Javascript, PHP)
- Databases: SQL
- Algebraic: Maude, CafeOBJ
- Modelling: UML
- WSC: BPEL
- Regex text processing: Perl
- Proof theory: **Coq**, Issabelle/HOL, Lean
- Programming language tools: Rascal, Spoofax, Racket, K
- ...

Debate

Which paradigm is better?

Indeed, this question is stupid...

Debate

Which paradigm is better?

Indeed, this question is stupid...

What's next?

- PL = Syntax + Semantics
- Syntax:
  - how can we combine symbols to create correct programs?
  - Grammars (BNF = Backus-Naur Form)
- Semantics: what programs *mean*?
- Other things related to PLs: pragmatics, compilers, etc.

What's next?

- PL = Syntax + Semantics
- Syntax:
  - how can we combine symbols to create correct programs?
  - Grammars (BNF = Backus-Naur Form)
- Semantics: what programs *mean*?
- Other things related to PLs: pragmatics, compilers, etc.

What's next?

- PL = Syntax + Semantics
- Syntax:
  - how can we combine symbols to create correct programs?
  - Grammars (BNF = Backus-Naur Form)
- Semantics: what programs *mean*?
- Other things related to PLs: pragmatics, compilers, etc.

What's next?

- PL = Syntax + Semantics
- Syntax:
  - how can we combine symbols to create correct programs?
  - Grammars (BNF = Backus-Naur Form)
- Semantics: what programs *mean*?
- Other things related to PLs: pragmatics, compilers, etc.

Questions?

Bibliography:

- *Chapter 13* from *Programming Languages: Principles and Paradigms*, Maurizio Gabbrielli, Simone Martini, Online: `http://websrv.dthu.edu.vn/attachments/newsevents/content2415/Programming_Languages_-_Principles_and_Paradigms_thereds1106.pdf`