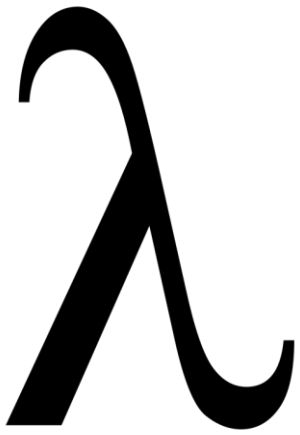



# Introduction

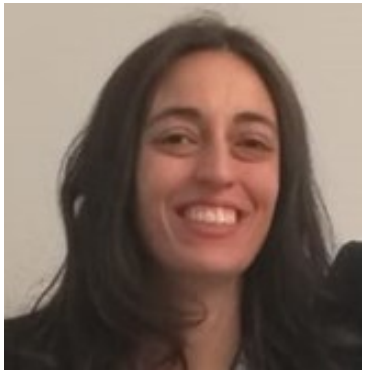
Programmazione Funzionale

2024/2025

Università di Trento

Chiara Di Francescomarino

# Lecturers



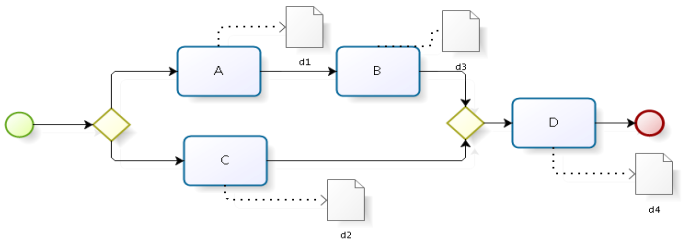
Chiara
Di Francescomarino

Sebastiano
Dissegna

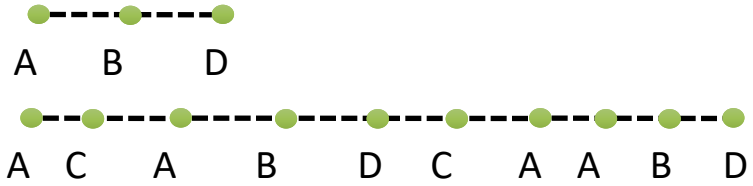
Tutor

# Who am I?

- Computer scientist with background in software engineering
- Currently working on




Business Process Models




Execution Traces

# Today


- What is the course about?
- What will we see in the course?
- What will we use in the course?
- Course organization
- A look at the history of programming languages

# What is the course about?

# A plethora of programming languages

- There exist many programming languages
- Some of them are similar but have a different syntax
- Some of them are based on completely different paradigms

# Why different programming languages?

- Historical reasons: new constructs, techniques, mechanisms
- Economical reasons: commercial products
- Different priorities: efficiency, flexibility, code readability
- Different usages: embedded systems, numerical computation, web applications, …

# Different characteristics

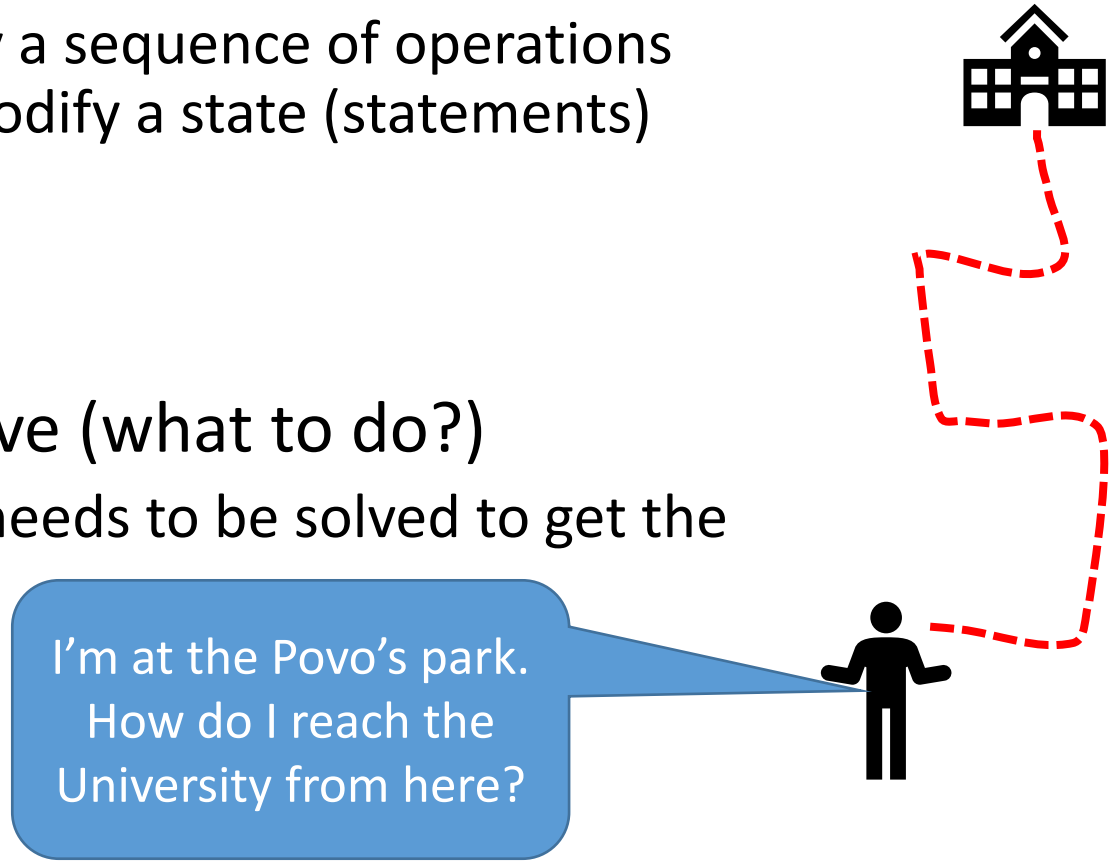
- Imperative (how to do?)
  - Specify a sequence of operations that modify a state (statements)

- Declarative (what to do?)
  - What needs to be solved to get the result

# Different characteristics …

- Imperative (how to do?)
  - Specify a sequence of operations that modify a state (statements)

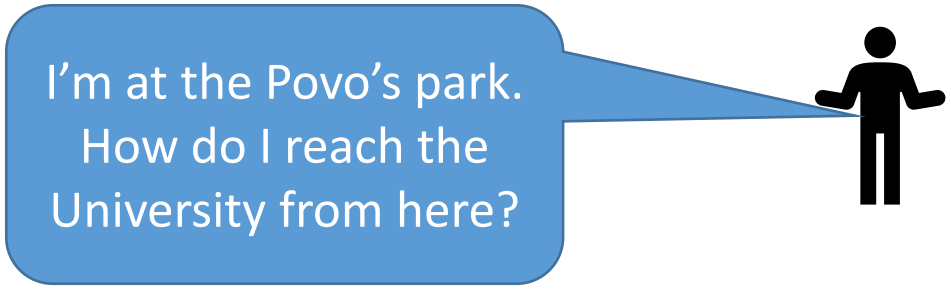
- Declarative (what to do?)
  - What needs to be solved to get the result



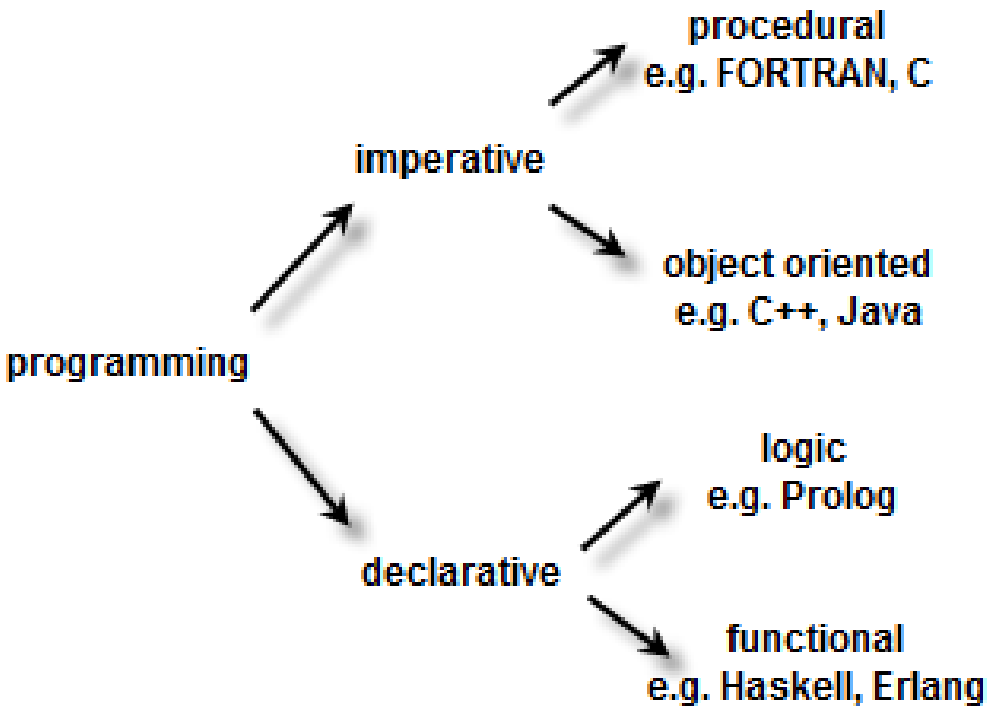
# Different characteristics …

- Imperative (how to do?)
  - Specify a sequence of operations that modify a state (statements)

- Declarative (what to do?)
  - What needs to be solved to get the result



# … different languages



# … different languages

- Imperative (how to do?)
  - Classical: Fortran, Pascal, C
  - Object-oriented: Smalltalk, C++, Java
  - Scripting: Perl, Python, Javascript
- Declarative (what to do?)

```
int main(){
     printf("Hello World");
     return 0;
}
```

```
public class HelloWorld{
  public static void
main(String[] args) {
     System.out.println("He
llo World"); }}
```

```
print ‘‘Hello, world!\n’’
```

# … different languages

- Imperative (how to do?)
  - Classical: Fortran, Pascal, C
  - Object-oriented: Smalltalk, C++, Java
  - Scripting: Perl, Python, Javascript

- Declarative (what to do?)
  - Logic: Prolog
  - Functional: ML, Ocaml

```
program (input, output)
```

```
output = program (input)
```

# What is functional programming?

- More a style than a paradigm
- You can write “functional code” in almost any language

# Some distinguishing features

- Recursion instead of iteration
- Pattern matching on values
- Expressions instead of statements
- Functions as first-class citizens

# Recursion instead of iteration

Sum integer numbers from 0 up to n

## Iteration (C)

- Repeating a process a number of times

```
int sumUpTo(int n) {
    int total = 0;
    for (int i = n; i >= 0; i--)
        total += i;
    return total;
}
```

## Recursion (ML)

- Defining something in terms of itself

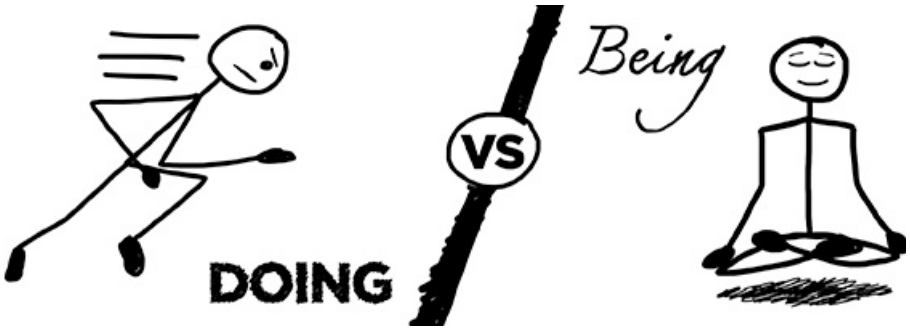
```
fun sumUpTo 0 = 0
      |sumUpTo n = n + sumUpTo(n-1)
```

# Pattern matching on values

- A function is defined by a series of equations
  - The value is compared with each left side until one “fits” (pattern matching)
  - In sumUpTo, if the value is zero, we return zero, otherwise we match the second one

```
fun sumUpTo 0 = 0
     |sumUpTo n = n + sumUpTo (n-1)
```

# Expressions instead of statements




What code does

- Statements manipulate the state of the program
- Statements have an inherent order
- Variables name and store pieces of state

What code is

- Value of a whole expression depends only on its subexpressions

```
sumUpTo 3 → 3 +sumUpTo 2
          → 3 + 2 + sumUpTo 1
          → ...
```

# Functions as first-class citizens

- Function: mapping of arguments to a result

```
fun greet name = "Hello, "^name^"!";
greet “Alice”
Hello, Alice!
```

- Functions can be parameters of another function
- Functions can be returned from functions

```
map greet [“Alice”,”Bob”]
[“Hello, Alice!”, “Hello, Bob!”]
```

where `map` applies the function `greet` to each element of the list

# Some consequences

- Programs are not executed but evaluated
- No side effects and no mutable state (no state)
  - A function has side-effects if it modifies some state in addition to producing a value

```
int calls = 0; // state
int sum(int a, int b) {
    calls++; // side-effect
    int tot = a + b ;
    printf("Total is %i.\n", tot); // side-effect
    return(tot); // actual result
}
```

# What is functional programming?

- Several definitions
- For sure … functional programming is not bound to a specific programming language
- Functional programming is a paradigm that can be applied using many different languages …

# What is functional programming?

- The key essence is writing programs without using side effects
  - Pure functions are functions without side effects
  - No mutable variables (variable assignments are side effects) and no mutable state
  - No loops but recursion

# Why functional programming?

- Functional programming introduces you to new ways to *think about* your programs:
  - new abstractions
  - new design patterns
  - new algorithms
  - elegant code
- Moreover:
  - functional programming techniques such as map-reduce offer a way to obtain efficient solutions
  - Concurrency for free (lack of side-effects)

# Why functional programming?

- Short term: fewer bugs
  - Types prevent errors
  - No side effects: a function cannot mutate a global state
  - Close to mathematics (proving properties)
- Long term: more maintainable
  - Higher-order functions remove a lot of boilerplate
  - Less code to test, compact code
  - Types help in refactoring

# FP is gaining traction

- Elements of functional programming are showing up all over:
  - F# in Microsoft Visual Studio
  - Scala combines ML (a functional language) with Objects
  - Python, Java 8, Javascript include “lambdas”
  - Javascript using functional programming techniques to write more elegant code
  - C++ libraries for map-reduce

# Some Functional Programming concepts

- Curried functions
- Type inferencing
- Polymorphism
- Higher-order functions
- Lambda expressions

# Do not be afraid!

- Functional programming languages can look a bit strange
- The elegant mathematic theory behind functional programming looks scary

  *“A monad is a monoid in the category of endofunctors, what’s the problem?”*

Don’t be afraid! It is not a solution for every problem but can be useful in many situations

# Is functional programming academic only?

- Strong mathematical foundations
  - $\lambda$-calculus, type theory
  - Monads, monoids, functions, endofunctors, …
- A lot of theoretical stuff
- Is it only good for writing papers?

# Not only …

WhatsApp ← Erlang

facebook ← Scala for correctness, maintainability and flexibility

Map-reduce in their data centers → Google

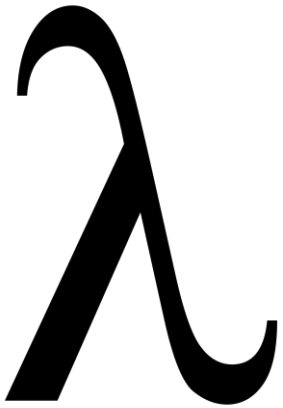
amazon ← Scala

Scala → pagoPA

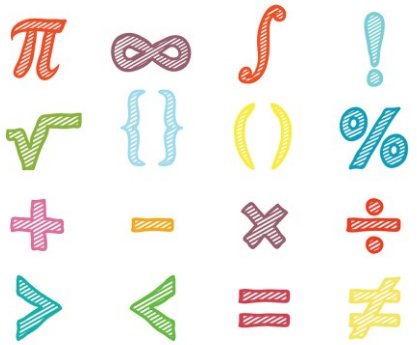
Haskell → 


- http://gregosuri.com/how-facebook-uses-erlang-for-real-time-chat
- http://labs.google.com/papers/mapreduce.html
- http://www.haskell.org/haskellwiki/Haskell_in_industry
- https://github.com/erkmos/haskell-companies
- https://github.com/pagopa


# What will we see in the course?



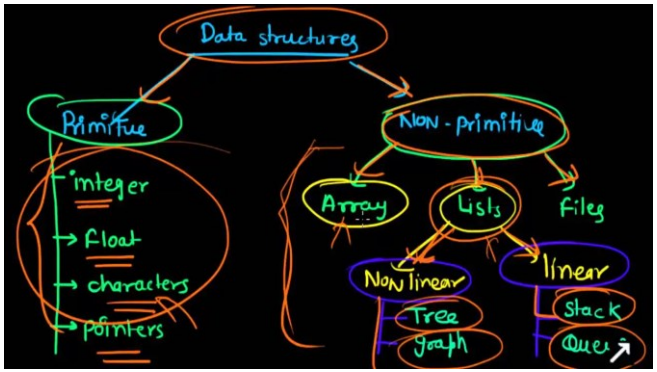
# Expressions and commands



- What is an expression?
- What is a command?
- What are the differences?
- What is a side effect?


Do we have both expressions and commands in functional programming?

# Data structures and type systems




- What kind of data types do high level languages use?
- How to deal with their compatibility and correctness?

What is static type checking? What is polymorphism?

# Names and environments


- What is a denotable object?
- What is the environment?
- What is static and dynamic binding?
- Where and when are the denotable objects visible?
  - Dynamic scoping
  - Static scoping

What kind of binding and scoping strategies are typically used with functional languages?

# Control structures and abstraction

- How do high-level languages deal with the abstraction of procedural data?
  - How to deal with their parameters?
  - How to deal with functions passed as parameters?

reference

value constant

value-result name

result

What is a call-by-name? What is a higher order function?

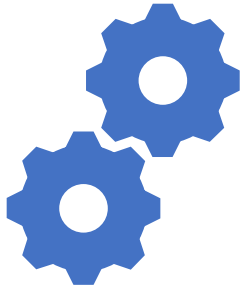
# Functional programming

- A functional programming language: ML

```
> fun sumUpTo 0 =0
      |sumUpTo n = n +sumUpTo (n-1)
```

- Lambda calculus
- An introduction to an object-oriented functional programming language: Scala

# Introduction to Logic Programming



file7.pro

10:34 Insert Indent

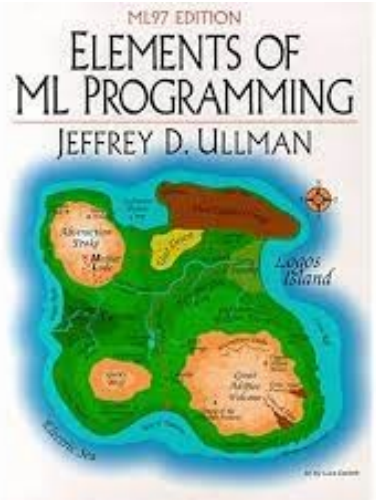
```
parent(person("Bill","male"),person("John","male")).
parent(person("Pam","female"),person("Bill","male")).
parent(person("Pam","female"),person("Jane","female")).
parent(person("Jane","female"),person("Joe","male")).

grandFather(Person,TheGrandFather):-
                    parent(Person,ParentOfPerson),
                    father(ParentOfPerson,TheGrandFather)
```



```
father(P, person(Name, "male")) :-
                parent(P,person(Name, "male")).
```

# Reference books

- Jeffrey D. Ullman, ”Elements of ML Programming ”, ML97 edition. Prentice-Hall
- Other material in Moodle



# Reference books

- Maurizio Gabbrielli and Simone Martini "Linguaggi di Programmazione - Principi e Paradigmi", McGraw-Hill
- Other material in Moodle


- Other books:
  - Scott, Michael L., “Programming language pragmatics", Morgan Kaufmann, 2019


# What will we use in the course?

# PolyML

- **PolyML:** http://polyml.org/index.html
- It is already installed on the lab machines
- You can find the instructions in Moodle to install it
  - For Windows: you can install 5.8.2 (https://github.com/polyml/polyml/releases/download/v5.8.2/PolyML5.8.2-64bit.msi)
  - For Linux: you can use apt:

    ```
    sudo apt install polyml
    ```
  - For Mac: find the file in Moodle

# Scala and Prolog

- **Scala 3**
  - It is already installed on the lab machines
    - You just need to install the Visual Studio plugins for syntax (scala-lang.scala) and Scala Metals (scalameta.metals)
  - If you want to install it: https://www.scala-lang.org/download/
- **SWI Prolog**
  - It is already installed on the lab machines
  - If you want to install it: https://www.swi-prolog.org/Download.html


# How is the course organized?

# Lectures

- Lectures on:
  - Tuesday 15:30 – 17:30 (A101)
  - Thursday 10:30 – 12:30 (Aula PC B107)

- Keep in mind that few lectures will be suspended
  - Thu 10/04 (TOLC) – probably we have to find another slot
  - Tue 15/04 (Provette)
  - Thu 01/05

- Tutoring: 1 hour per week – slot to be chosen

# In practice

- We will try to alternate
  - Theoretical classes on the main concepts of functional programming and ML constructs
  - Practical labs on exercises in ML

- For the reception: contact me via email (c.difrancescomarino@unitn.it)

# Final Exam

- In two parts
  - Multiple choice exam on the topics of the course (50%). Passing this part is required to take the second part.
  - Programming problem(s) in ML. (50%).
- Moreover, we will have a group challenge giving you an extra point

TEST

# PolyML for the exam

- It is already installed on PCs in lab
- For the exam, please try the version in the lab, and make sure you can create a file and save it

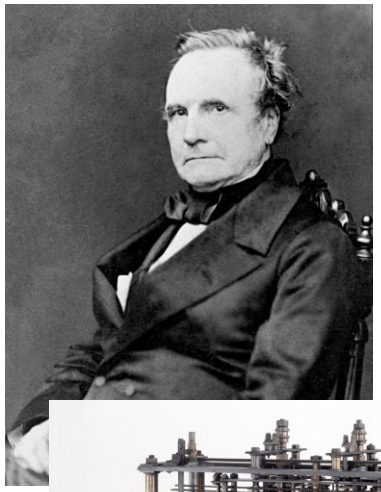
TEST

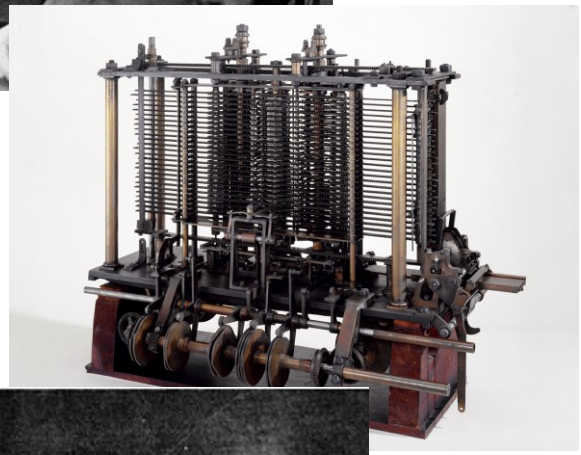

Questions?

# A bit of history of the programming languages

# Early history (1837)

- Charles Babbage developed the analytical engine – a mechanical general-purpose computer
- Ada Lovelace wrote the first program to be executed by the machine for computing the Bernoulli's number … the first program of the history



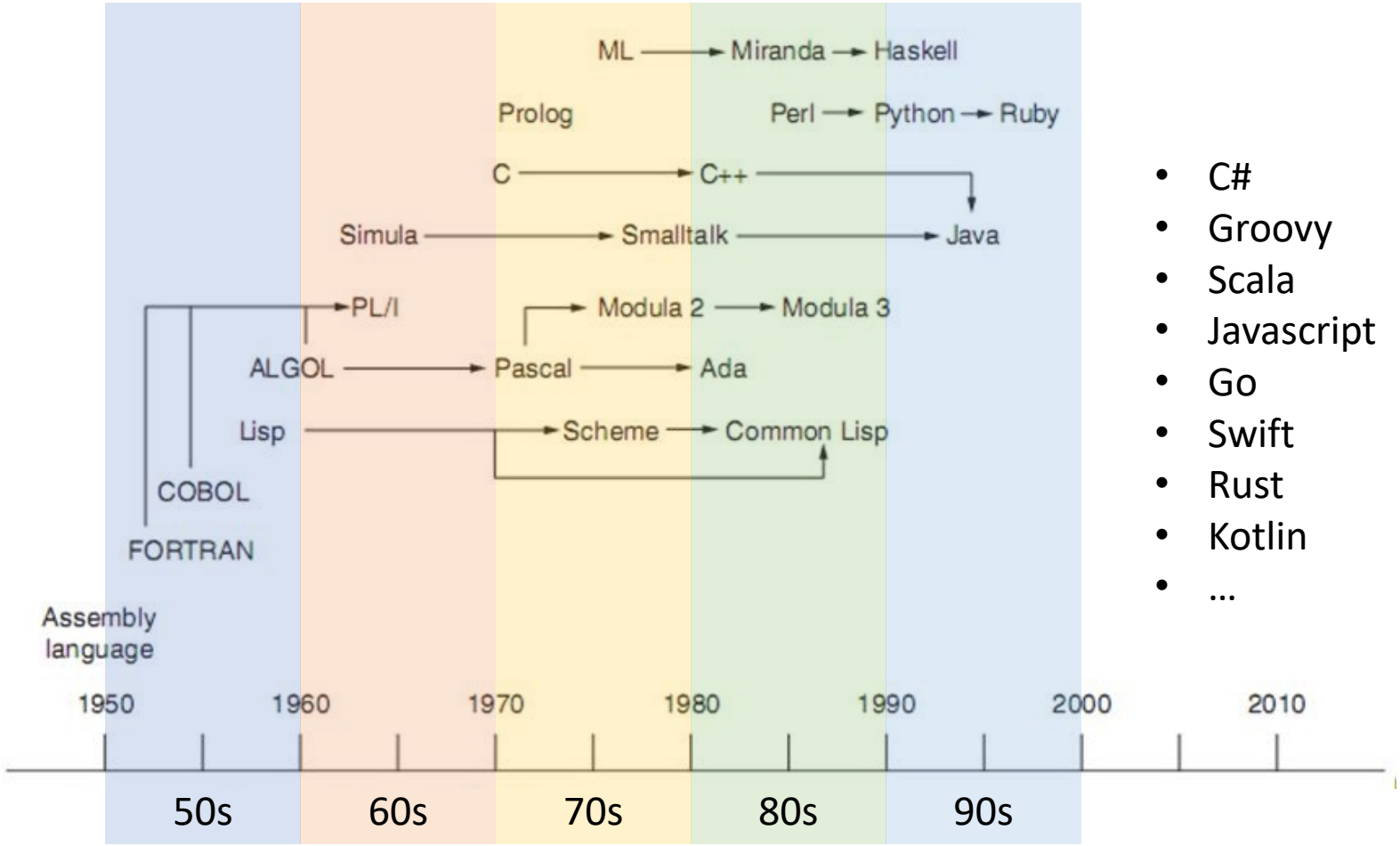


# Early computers and machine language (40s)

- Early computers
  - With von Neumann storable programs
  - Programmed in raw machine codes.
- Entirely numeric:
  - Poor readability
  - Poor modifiability
  - Expensive and tedious coding

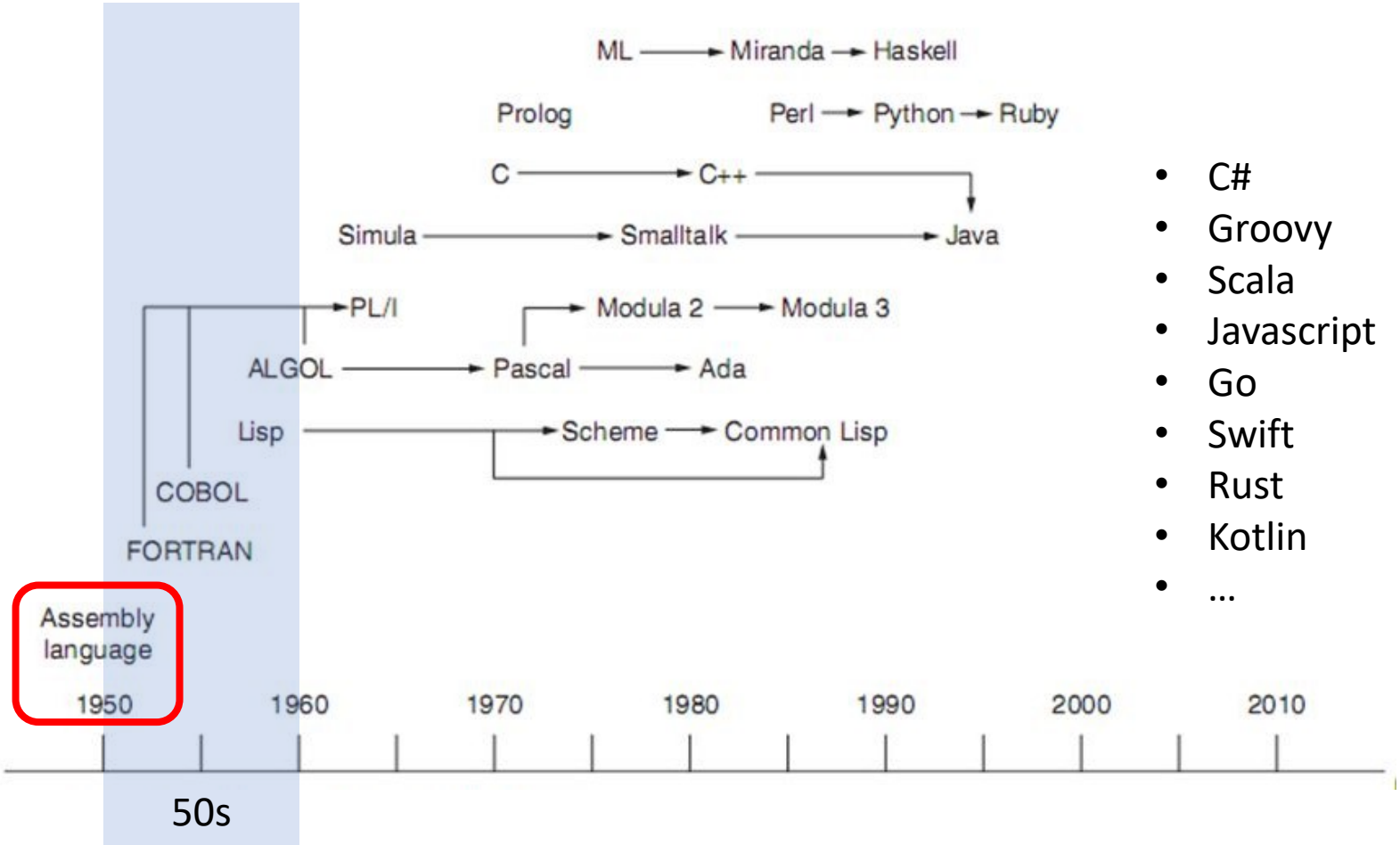
```
00011111000010101001
00111001110011010100
01010101010100000000
10101010101010101010
10101000001111100000
10101010001110001010
10101010010100100100
```

# Programming languages timeline




- C#
- Groovy
- Scala
- Javascript
- Go
- Swift
- Rust
- Kotlin
- …

# Programming languages timeline



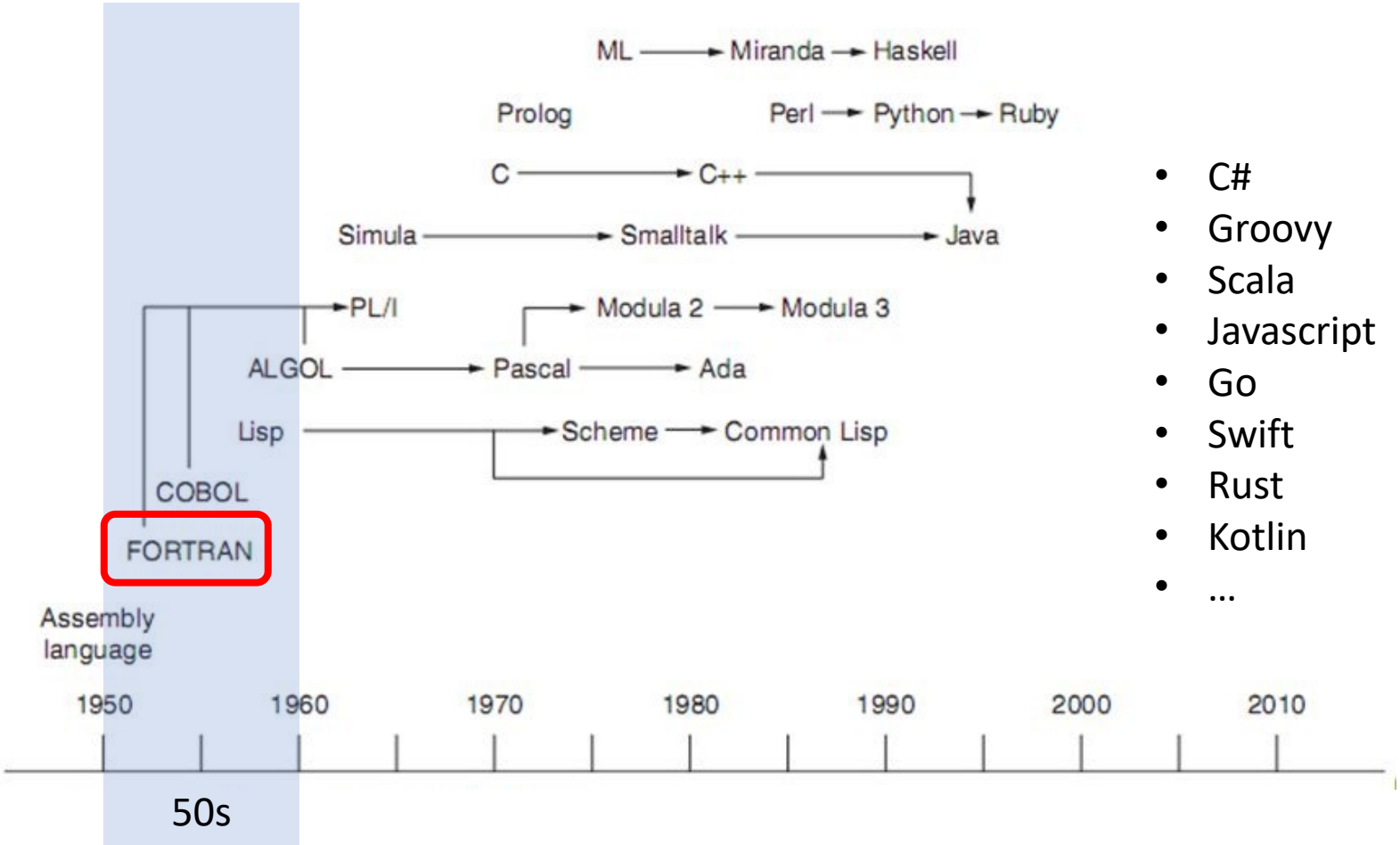

- C#
- Groovy
- Scala
- Javascript
- Go
- Swift
- Rust
- Kotlin
- …

# Assembly language (1949)

- Symbolic representations of the machine language (2nd generation)
- More readable names for machine instructions
- Translated into programs written in machine language by a program called an assembler.
- Their portability is very low.

```
; Example 2: Assembly Language
;
EXAMPLE2: MOV  DPTR,#50H       ;init pointer to 0050H
          MOV  R7,#0           ;init count = 0
REPEAT:   MOVX A,@DPTR         ;char = @pointer
          INC  DPTR            ;increment pointer
IF:       CJNE A,#'0',$+3      ;if char >= '0' AND
          JC   UNTIL           ;
          CJNE A,#'9'+1,$+3    ; char <= '9'
          JNC  UNTIL           ;
THEN:     INC  R7              ;then increment counter
UNTIL:    CJNE A,#0,REPEAT     ;char is 00H
          MOV  A,R7            ;store count in acc
HERE:     SJMP HERE
          END                  ;example 2
```

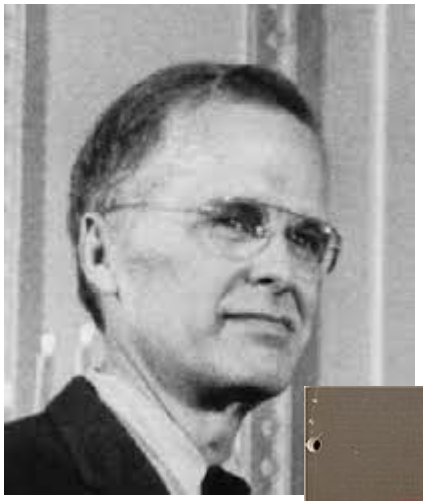
# Programming languages timeline

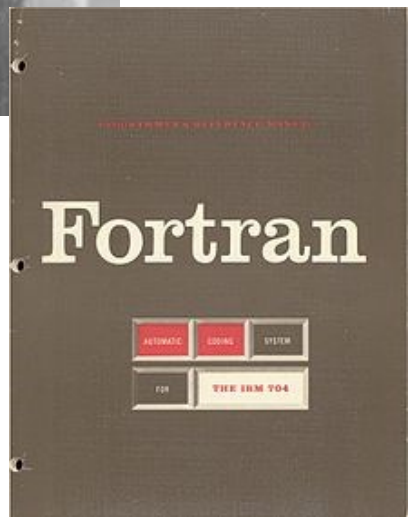



- C#
- Groovy
- Scala
- Javascript
- Go
- Swift
- Rust
- Kotlin
- …

# FORTRAN (50s)

- FORmula TRANslation language
- The first high-level language (3rd generation)
- Developed by John Backus and IBM
- Designed for applications of numerical-scientific type
- Procedural language



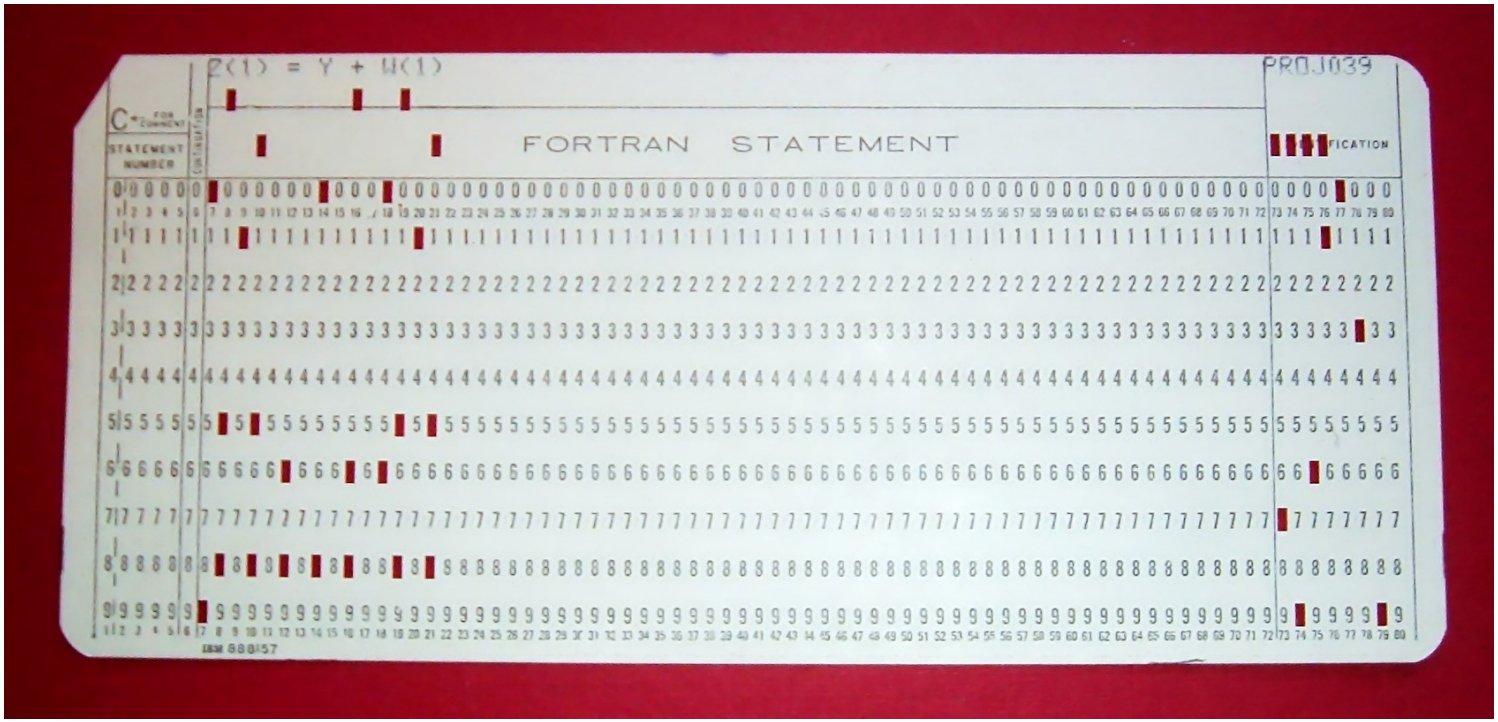


# FORTRAN characteristics

- PROS
  - Very good for floating point and algebraic notation
  - Introduces variables and arrays (fixed size)
  - Procedures, that can be compiled independently
  - Local and global environments
  - FORMAT for I/O
- CONS
  - Limited structured control sequence (mainly goto)
  - Limited data support
  - Procedures, but no recursion – memory statically allocated.

# Input

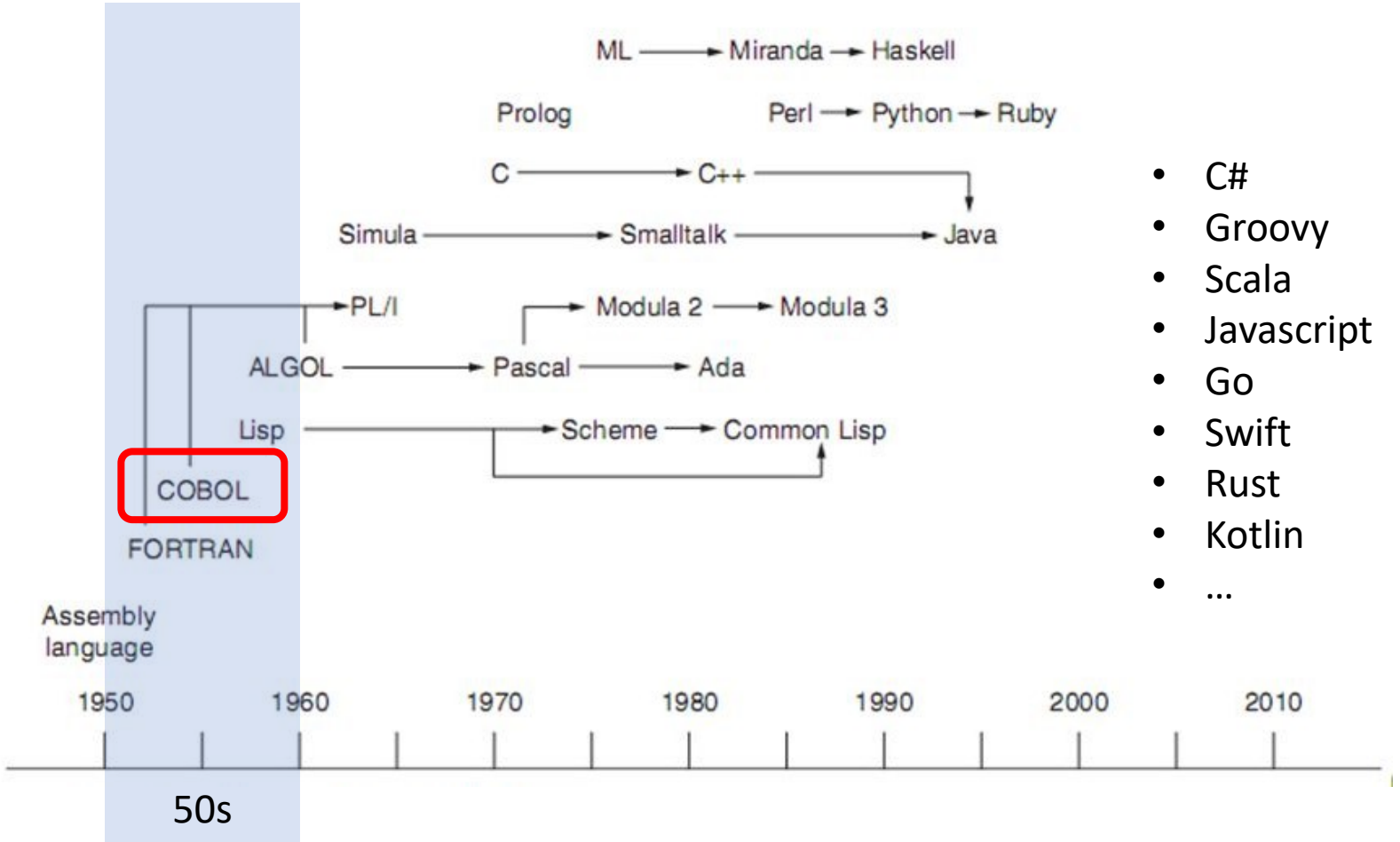
- Programs usually run in batch mode
- Input: stack of punched cards






Picture by Arnold Reinhold

# Programming languages timeline

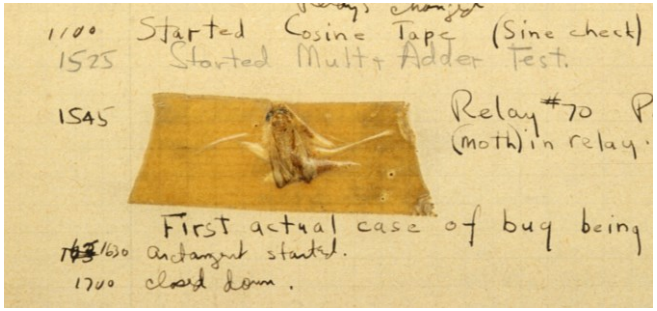



- C#
- Groovy
- Scala
- Javascript
- Go
- Swift
- Rust
- Kotlin
- …

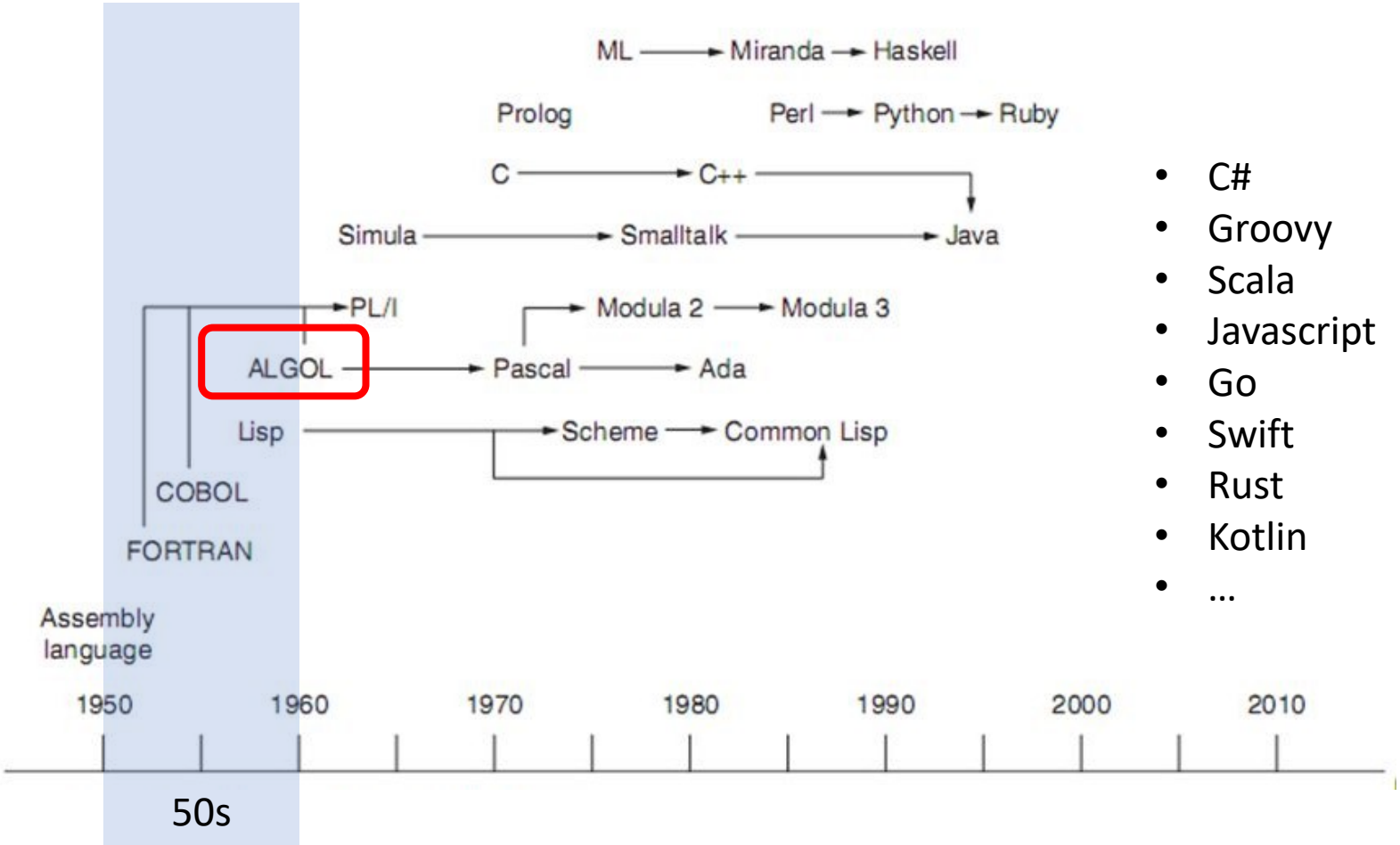
# COBOL (50s/60s)

- Common Business Oriented Language
- Designed by a team lead by Grace Hopper at the US Department of Defense
- Block structure BEGIN,END
- Introduces the RECORD type (in C, struct)
- Focus on data
- Static memory management
- Very verbose: the intent was to program in natural language


```
000001 IDENTIFICATION DIVISION.
000002 PROGRAM-ID.     HELLOWORLD.
000003 ENVIRONMENT DIVISION.
000004 CONFIGURATION SECTION.
000005 DATA DIVISION.
000006 PROCEDURE DIVISION.
000007
000008     DISPLAY 'HELLO, WORLD.'.
000009     STOP RUN.
```



# Programming languages timeline




- C#
- Groovy
- Scala
- Javascript
- Go
- Swift
- Rust
- Kotlin
- …

# ALGOL (50s/60s)

- ALGOrithmic Language
- Not designed for a particular application domain
- Introduced by a joint committee of **American** (ACM) and **European** (GAMM) experts
- Family of imperative languages. Three main versions
  - 1958
  - 1960
  - 1968

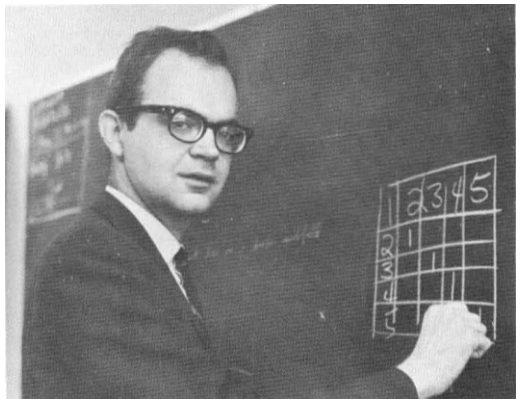
# ALGOL … further novelties

- ALGOL(60) directly influenced most imperative languages
  - Introduction of the BNF syntax
  - Block structure, with scope for local variables
  - Explicit type declarations
  - Recursion
  - Call by name
  - If-then-else
- Key point: Designed around a model of implementation using a stack

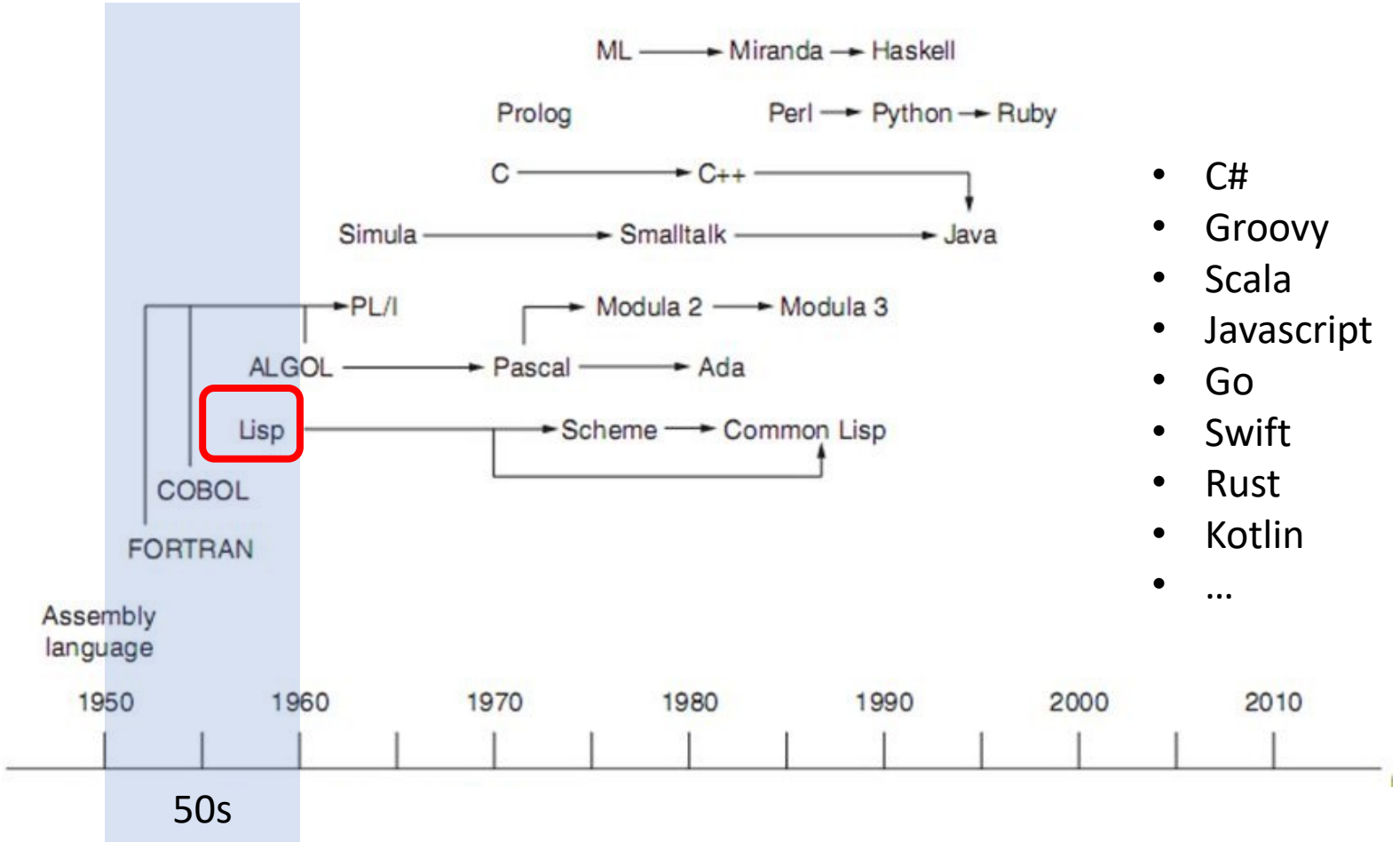
# Backus-Naur Form

```
<symbol> ::= __expression__
```

- John Backus developed the Backus Normal Form method of describing programming languages syntax specifically for ALGOL 58.
- It was revised and expanded by Peter Naur for ALGOL 60, and at Donald Knuth's suggestion renamed in Backus–Naur Form.
- The Backus-Naur Form has been used ever since then in the design of programming languages.



# Programming languages timeline



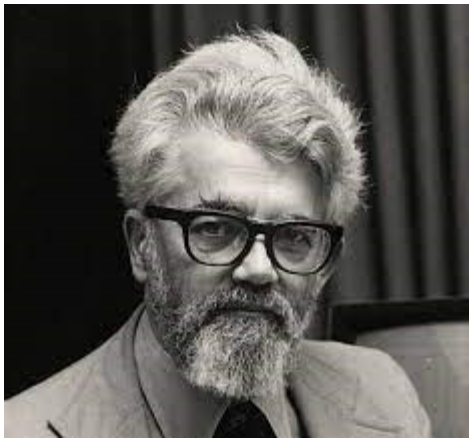

- C#
- Groovy
- Scala
- Javascript
- Go
- Swift
- Rust
- Kotlin
- …

# LISP (60s)

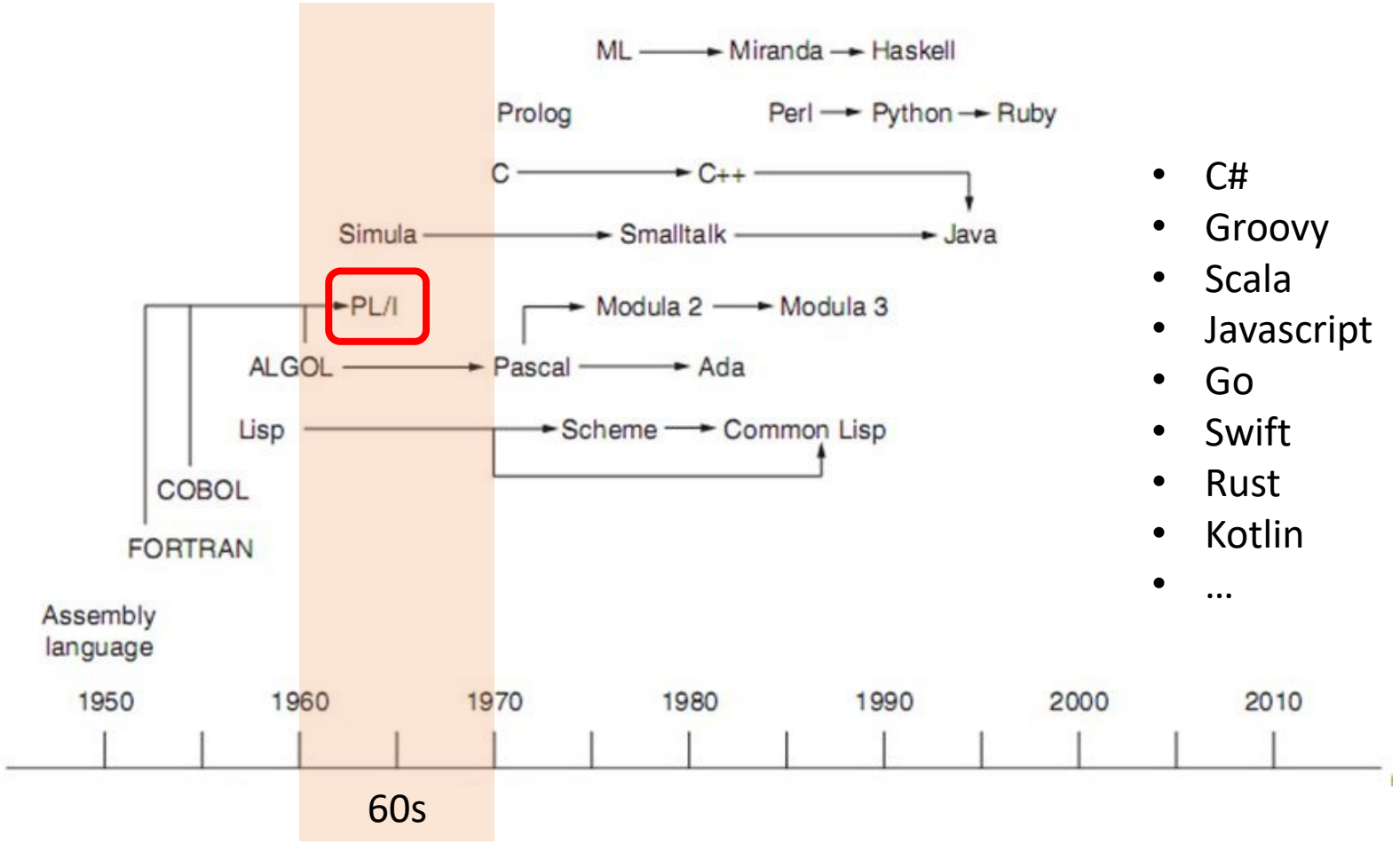
- LISt Processing
- Created by John McCarthy and based on the lambda-calculus by Alonzo Church at MIT
- Designed for AI: for manipulating s-expressions (symbolic expressions)
- Not a pure functional system, but a step in this direction
- Key feature: List processing

```
(defun foo (a b c d) (+ a b c d))
```

- Functions can be used as parameters and results of other functions
- First implementations were inefficient -> solved with dynamic management of memory using a heap and garbage collection

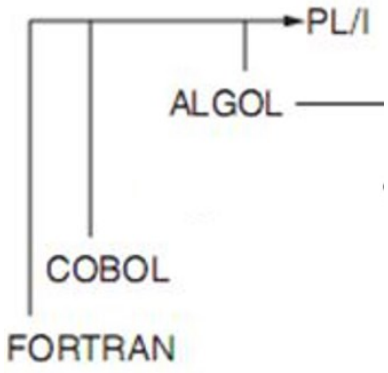


# Programming languages timeline



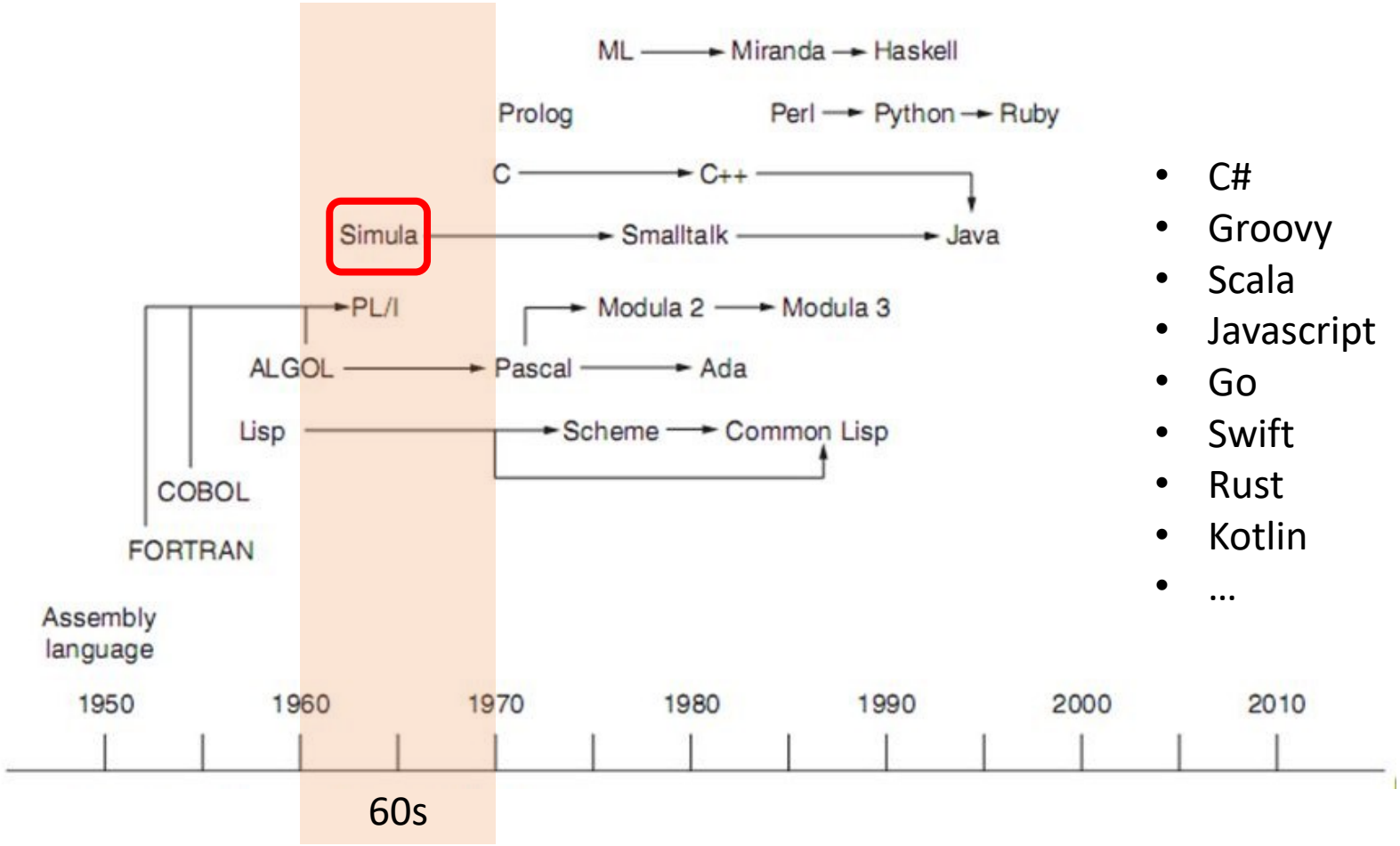

- C#
- Groovy
- Scala
- Javascript
- Go
- Swift
- Rust
- Kotlin
- …

# PL/I (60s)




- Programming Language 1
- Fortran emphasized floating point arithmetic, arrays, procedures, fast computation. Cobol emphasized decimal arithmetic, fast asynchronous input/output, string handling, efficient search/sort routines.
- Synthesis of previous languages all these features incorporated in a single language
- New:
  - Exception handling
  - Pointers

# Programming languages timeline

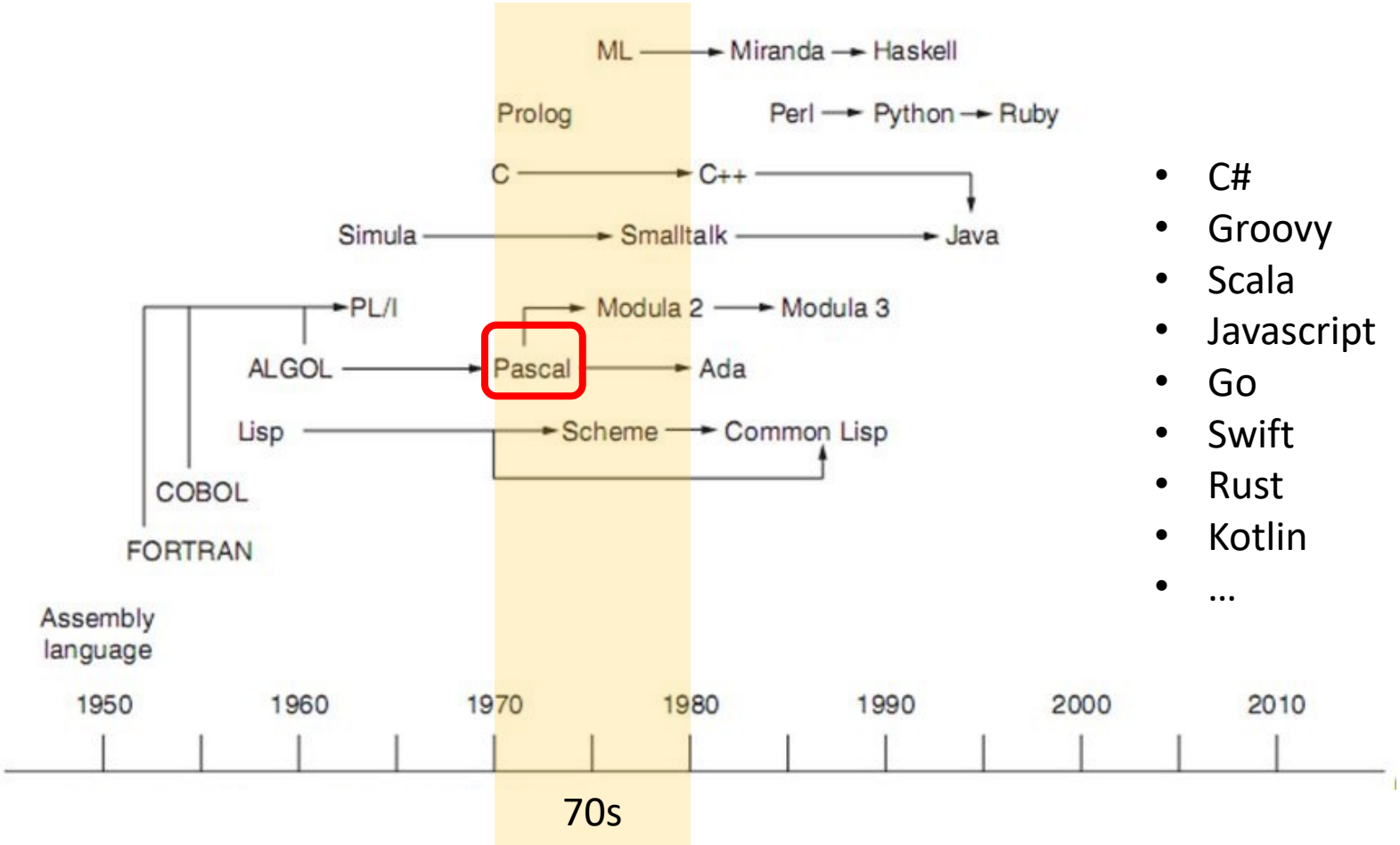



- C#
- Groovy
- Scala
- Javascript
- Go
- Swift
- Rust
- Kotlin
- …

# Simula (60s)

- An extension of ALGOL60
- Developed at the Norwegian Computing Center in Oslo, by Ole-Johan Dahl and Kristen Nygaard
- Designed for discrete-event simulation applications (loads and queues)
- The first object-oriented programming language
- It introduces the notion of objects, classes, subtype, garbage collection

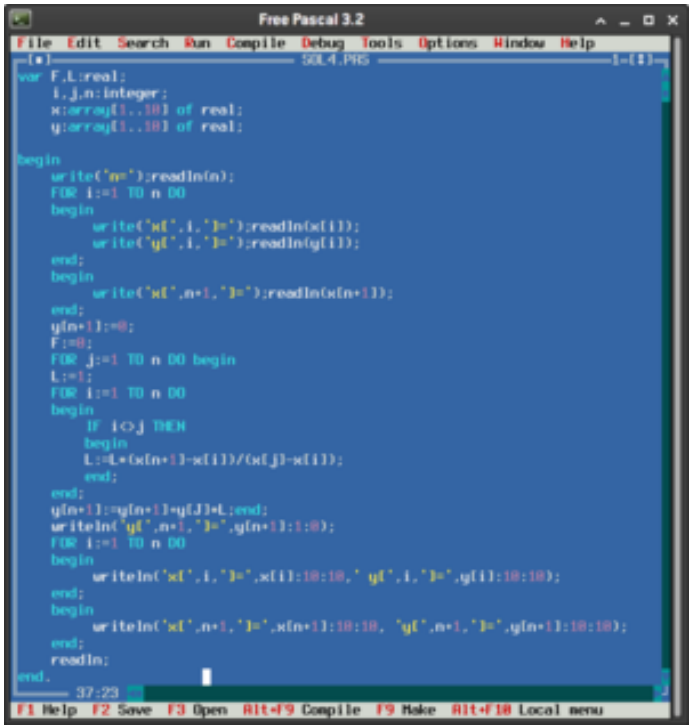
# Programming languages timeline

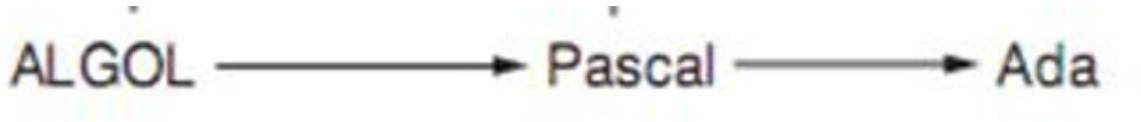



- C#
- Groovy
- Scala
- Javascript
- Go
- Swift
- Rust
- Kotlin
- …

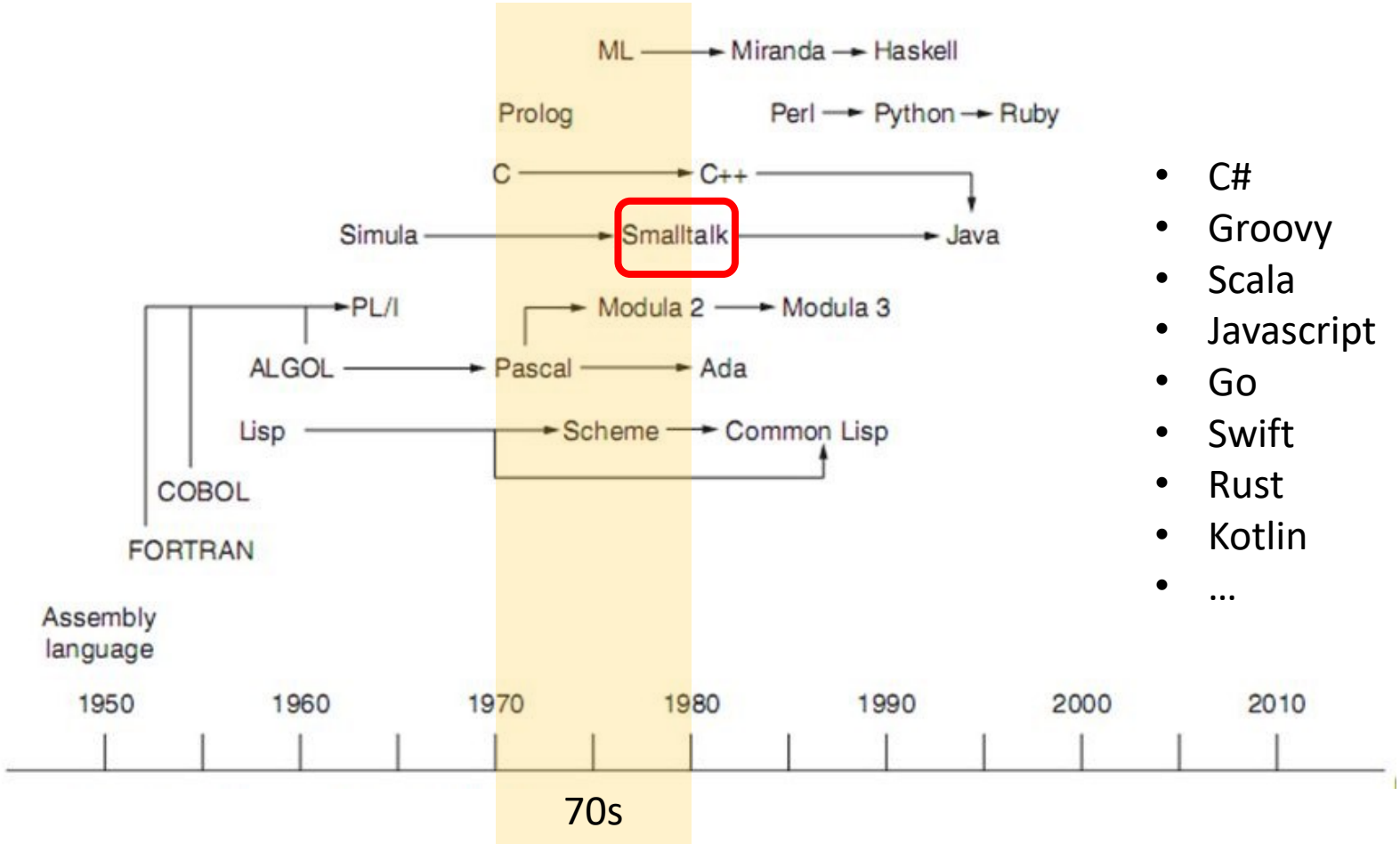
# Pascal (70s)

- An extension of ALGOLW
- Designed by Niklaus Wirth
- Very successful for education
- First language introducing intermediate code
- Strong typing
- Introduces the case statement



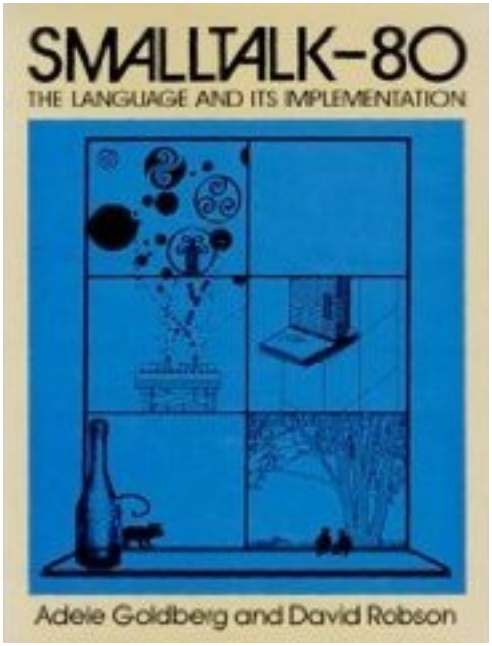
# Programming languages timeline




- C#
- Groovy
- Scala
- Javascript
- Go
- Swift
- Rust
- Kotlin
- …

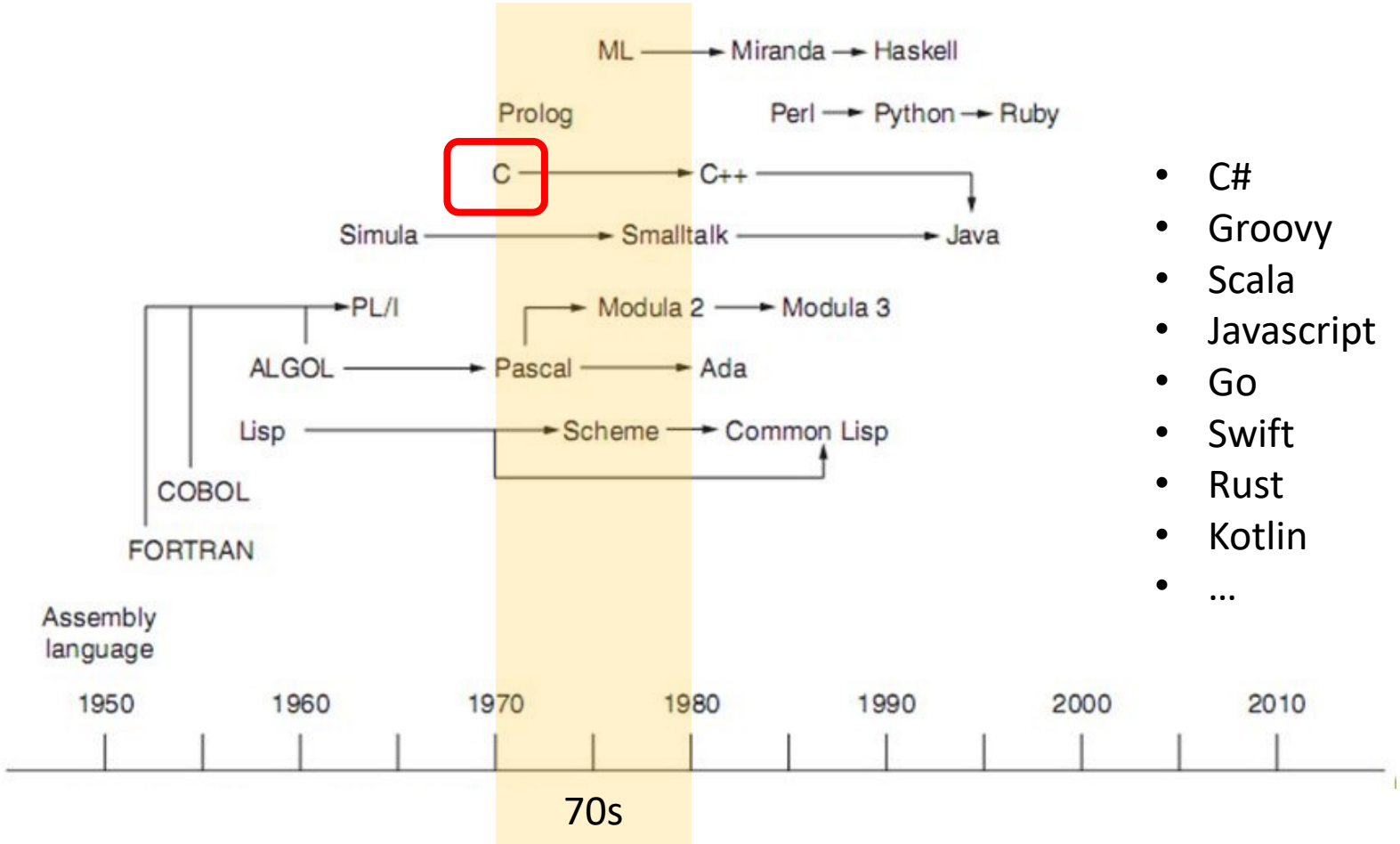
# Smalltalk (70s)

- Designed at the XEROX PARC by Alain Key
- Encapsulation and information hiding
- Fully object-oriented language (even constants are objects)
- Pioneer of extreme programming




Simula → Smalltalk → Java

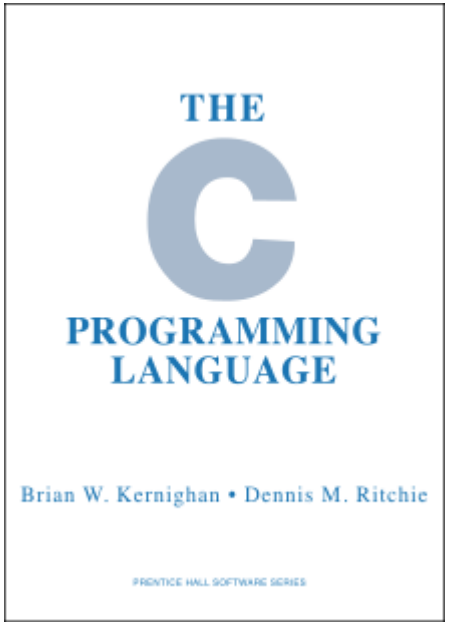
# Programming languages timeline



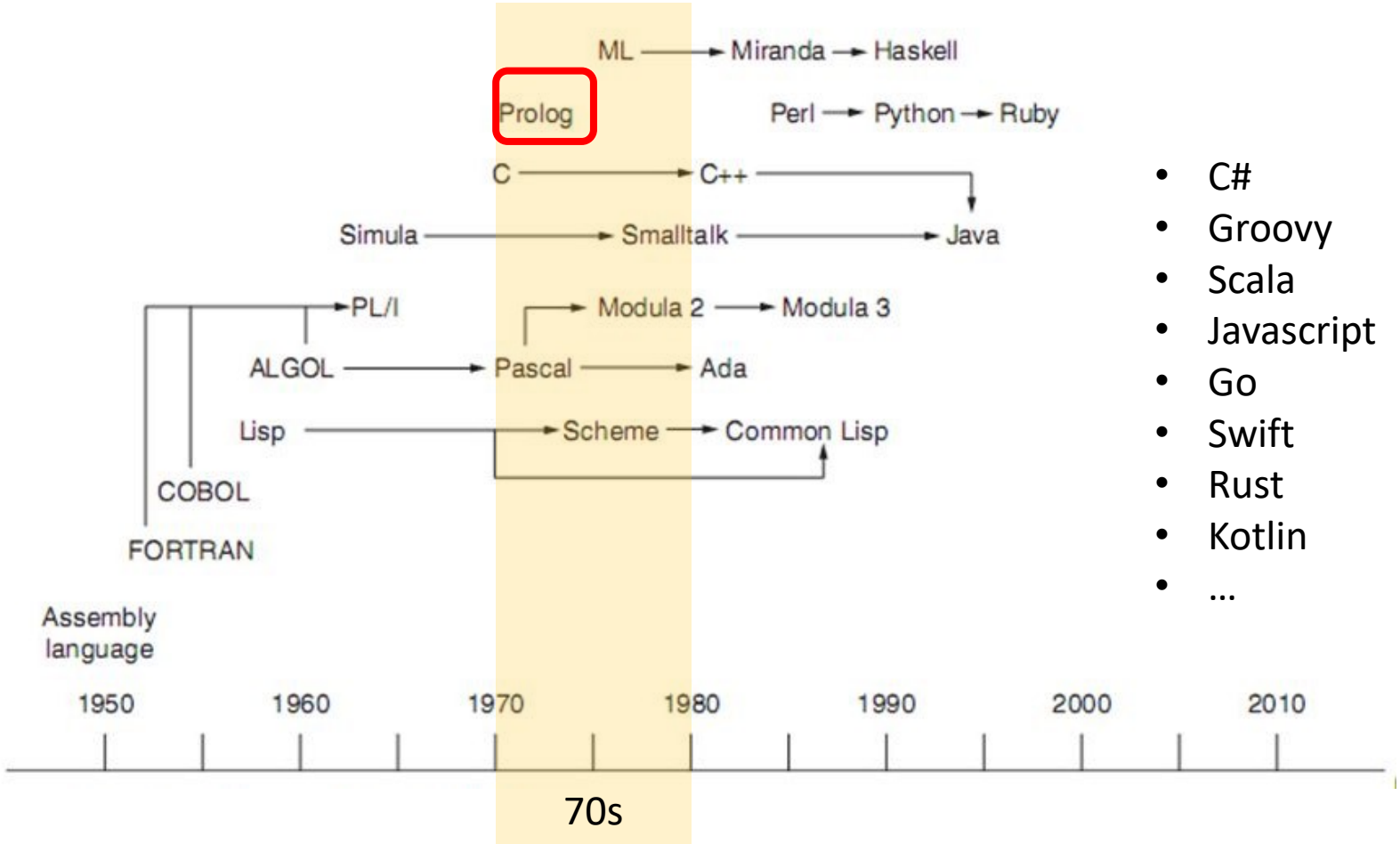

- C#
- Groovy
- Scala
- Javascript
- Go
- Swift
- Rust
- Kotlin
- …

# C (70s)

- Designed by Dennis Ritchie and Ken Thompson at the Bell Telephone Laboratories for system programming (especially Unix operating system)
- The successor of a language called B (Basic Combined Programming Language)
- Has constructs that map efficiently to machine instructions
- C does not allow nested functions
- CONS: Lack of a strong type system and pointers that can be manipulated



# Programming languages timeline



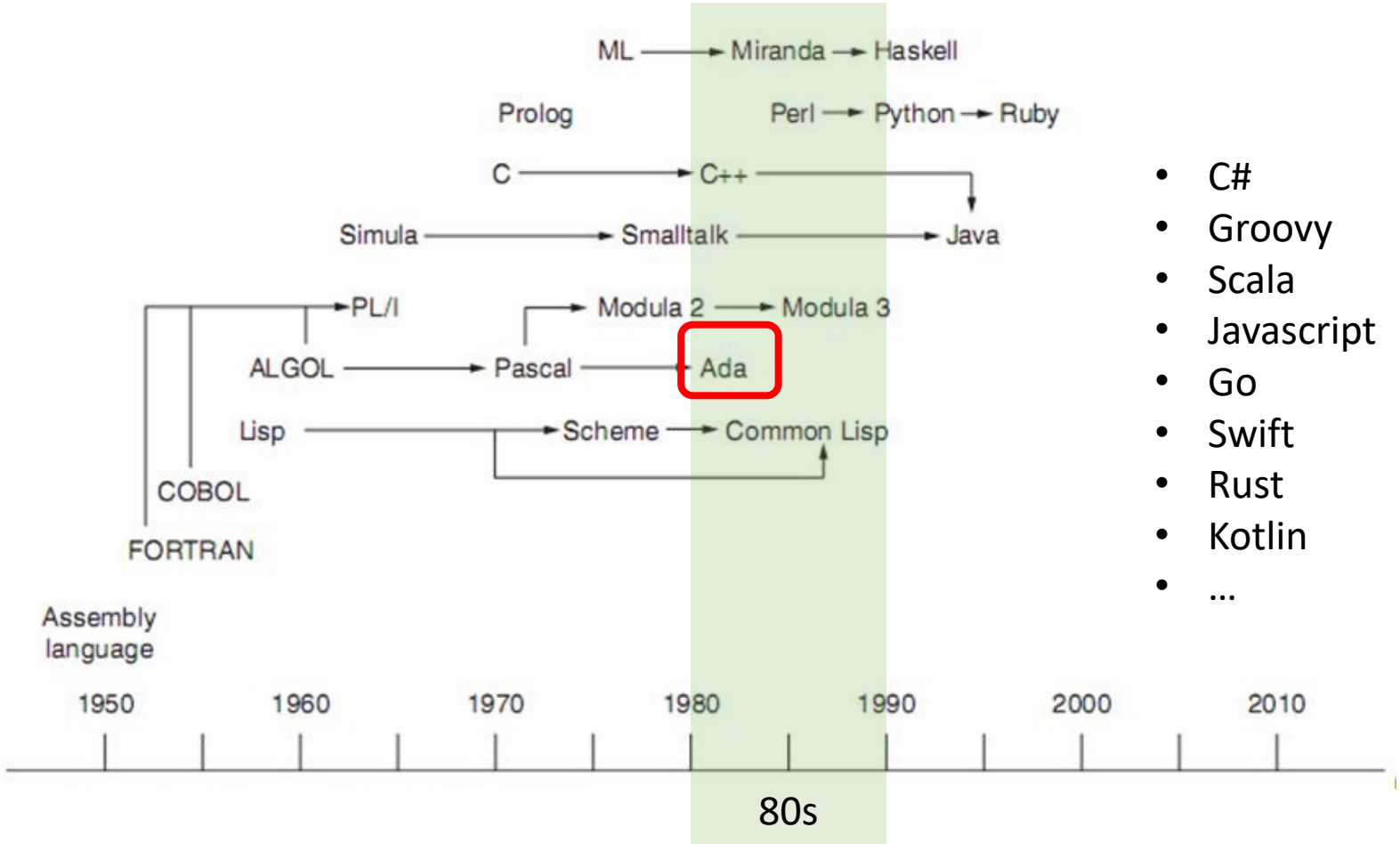

- C#
- Groovy
- Scala
- Javascript
- Go
- Swift
- Rust
- Kotlin
- …

# Prolog (70s)

- Developed at the University of Aix Marseille, by Colmerauer and Roussel, with some help from Kowalski at the University of Edinburgh
- Based on formal logic
- Non-procedural: write programs in logic
- Declarative language expressing knowledge in terms of facts and rules
- Uses an inferencing process to infer the truth of given queries (Inference done automatically)

```
cat(tom).
animal(X):-cat(X).
?- animal(X).
X = tom
```

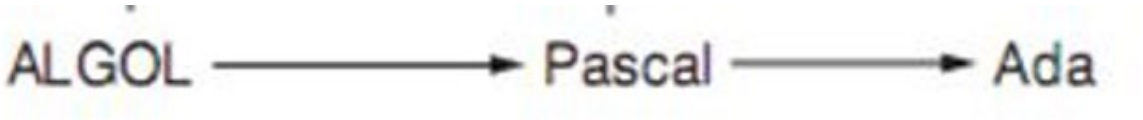
# Programming languages timeline



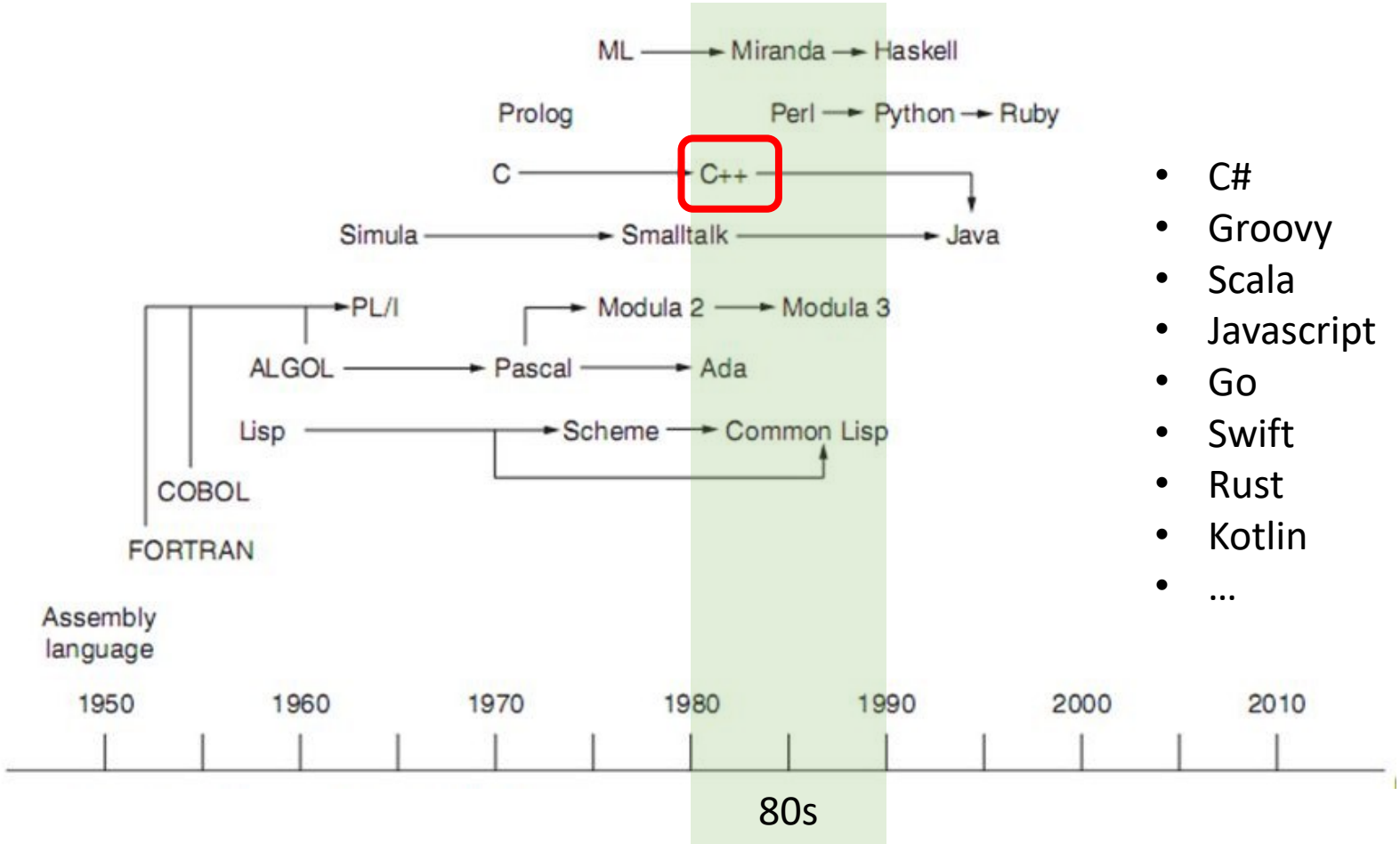

- C#
- Groovy
- Scala
- Javascript
- Go
- Swift
- Rust
- Kotlin
- …

# ADA (80s)

- Designed by a team led by Jean Ichbiah of CII Honeywell Bull in France under contract to the United States Department of Defense for embedded and real-time systems
- In the ALGOL tradition
- Very strong typing, packages, run-time checking and concurrency
- Modular programming support

# Programming languages timeline



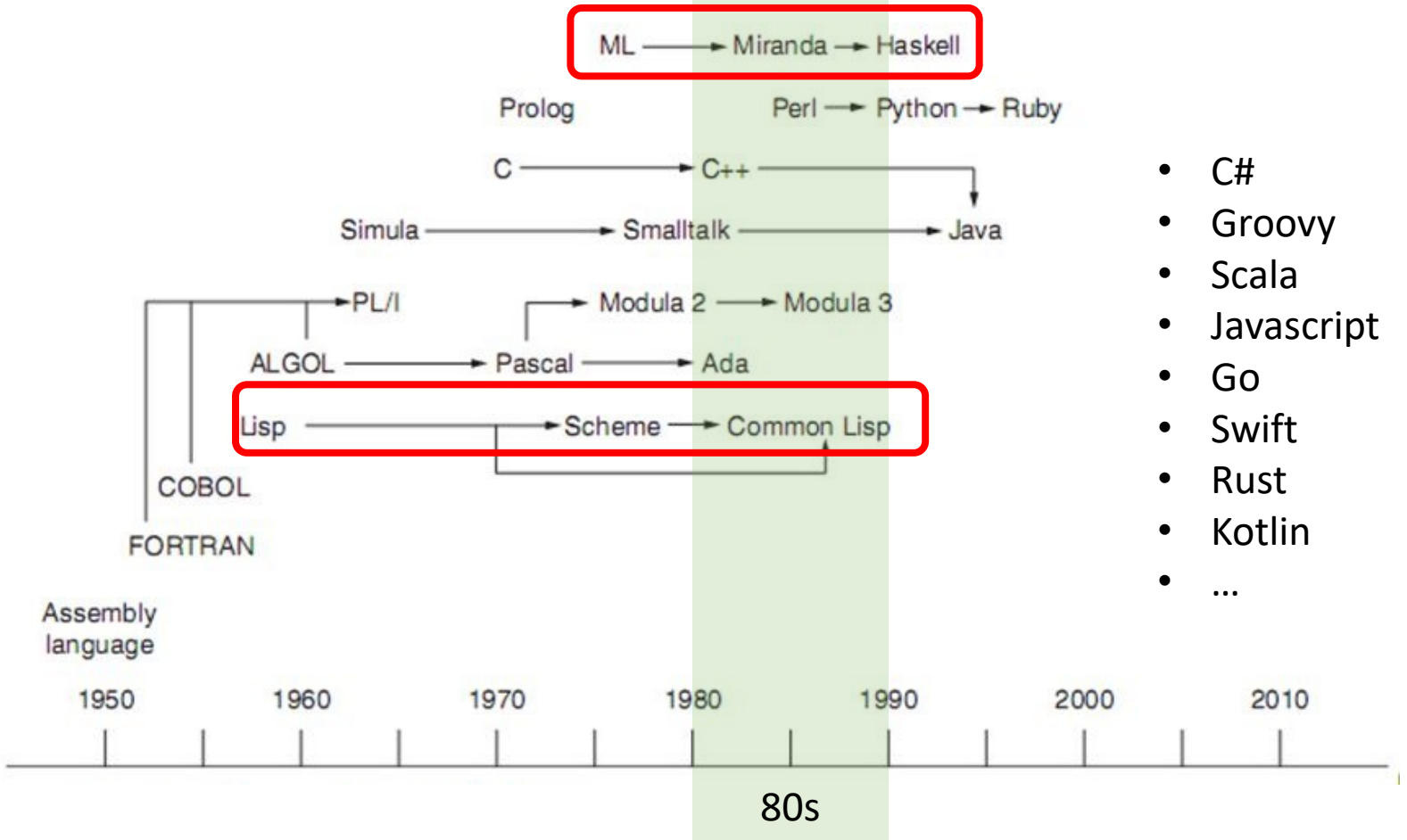

- C#
- Groovy
- Scala
- Javascript
- Go
- Swift
- Rust
- Kotlin
- …

# C++ (80s)

C ⟶ C++

- Developed by Bjarne Stroustrup at Bell Laboratories as an enhancement to the C programming language following the object-oriented principles pioneered by Simula.
- A combination of both high-level and low-level language features.
- Add the notion of classes and inheritance to C

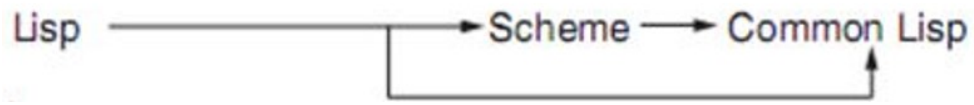
# Programming languages timeline

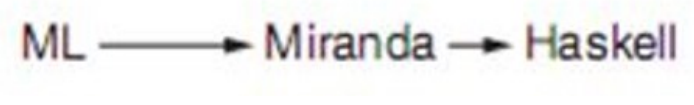



- C#
- Groovy
- Scala
- Javascript
- Go
- Swift
- Rust
- Kotlin
- …

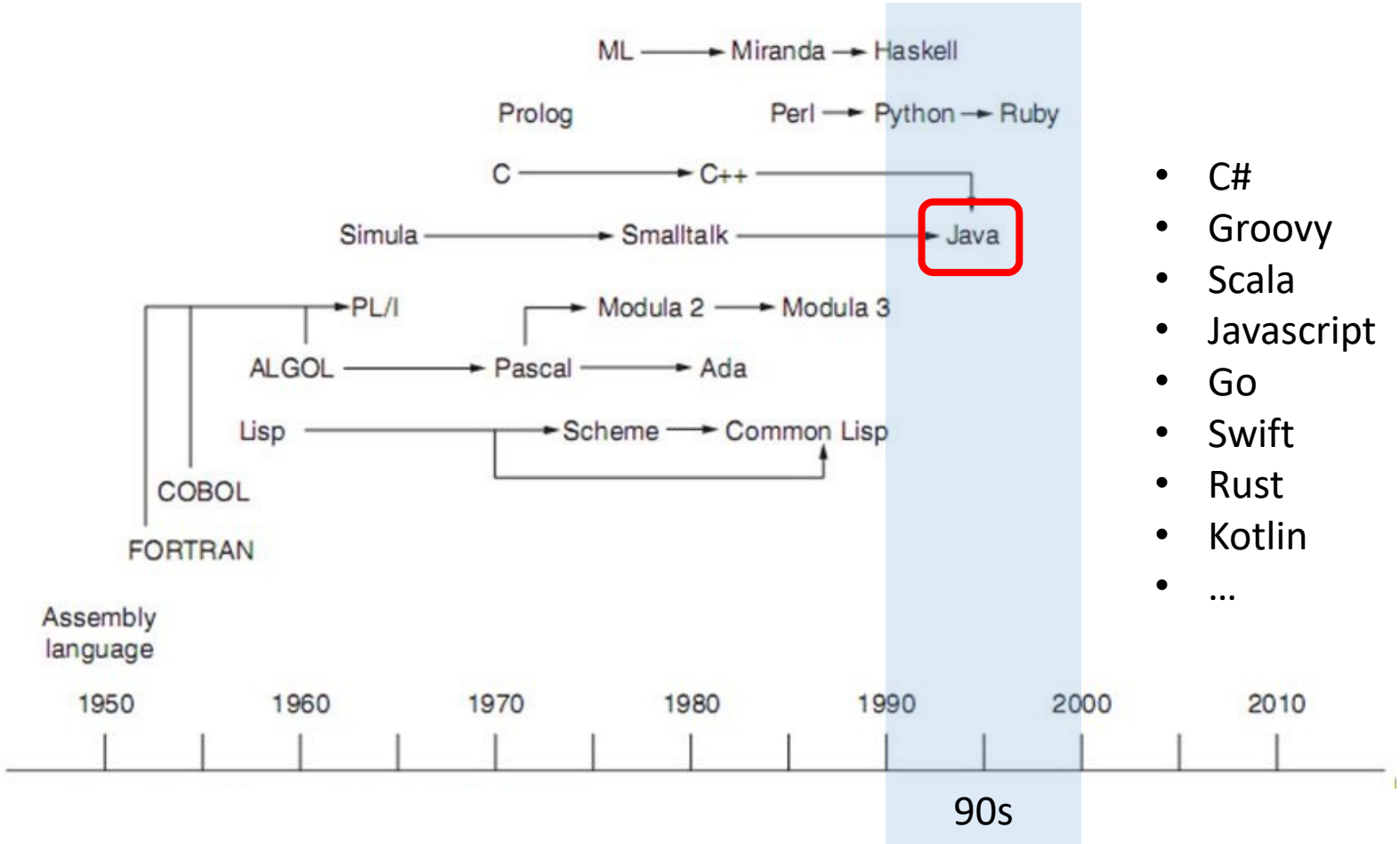
# Functional Programming (80s)

- Subject of this course
- Some aspects of functional programming in LISP and other languages
- Other languages: Scheme, OCaml, Haskell, Miranda, ML
- Miranda and Haskell are purely functional. Most others allow some side-effects.



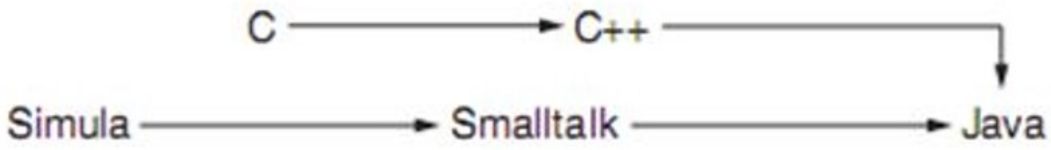



# Programming languages timeline



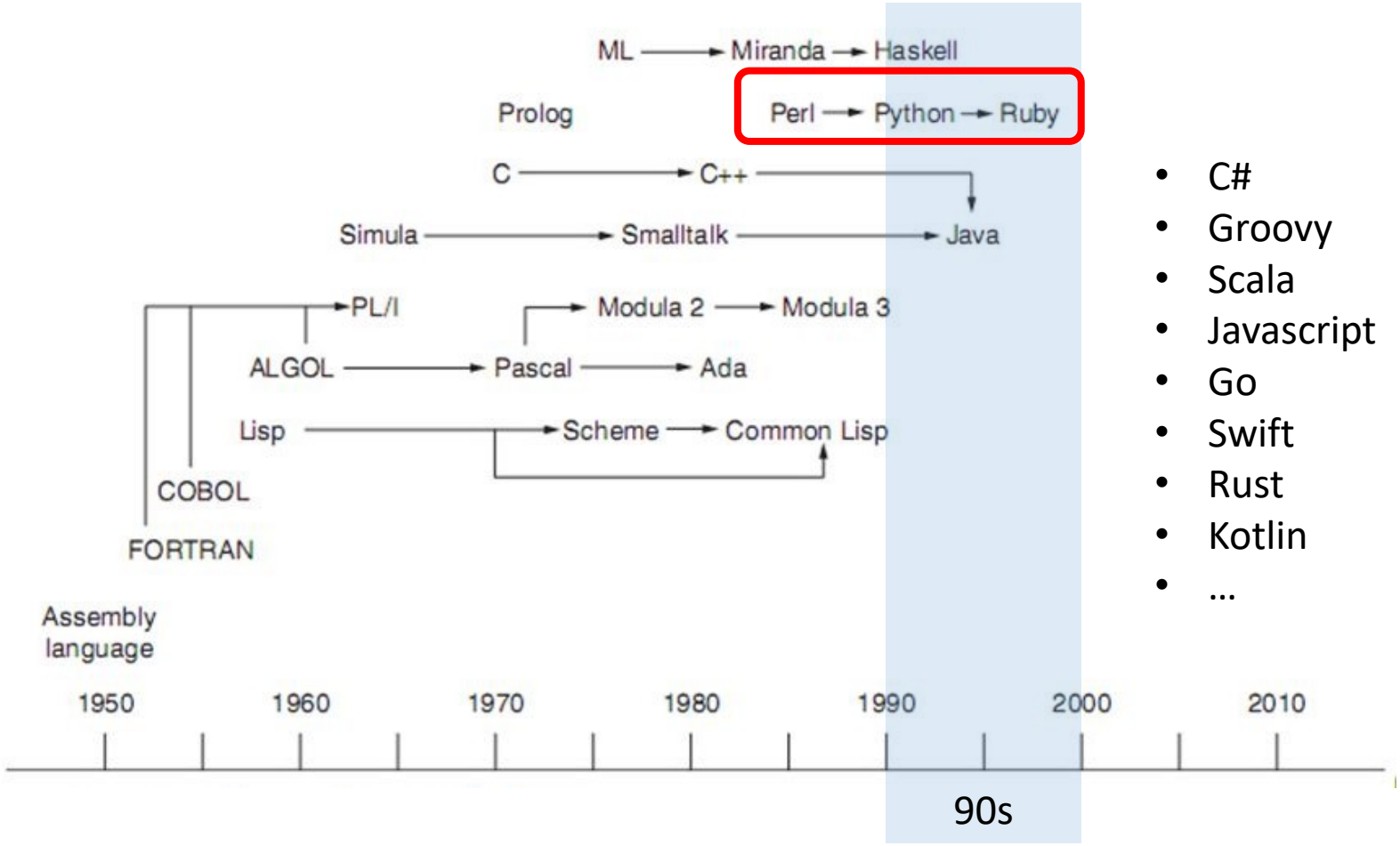

- C#
- Groovy
- Scala
- Javascript
- Go
- Swift
- Rust
- Kotlin
- …

# Java (90s)



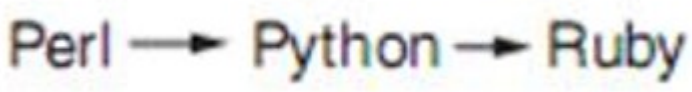

- Developed by Gosling at SUN Microsystems
- Object-oriented language, designed to be fully portable and secure
- Portability
  - Programs compiled into “bytecode" that can be run on any Java Virtual Machine
  - Interpreted bytecode + better performance with “just-in-time" compilers
- Security: type safety at three levels
  - Java compiler
  - Bytecode typechecker
  - Bytecode interpreter performs some checks
- Originally for embedded systems, now used widely on the Internet

# Programming languages timeline

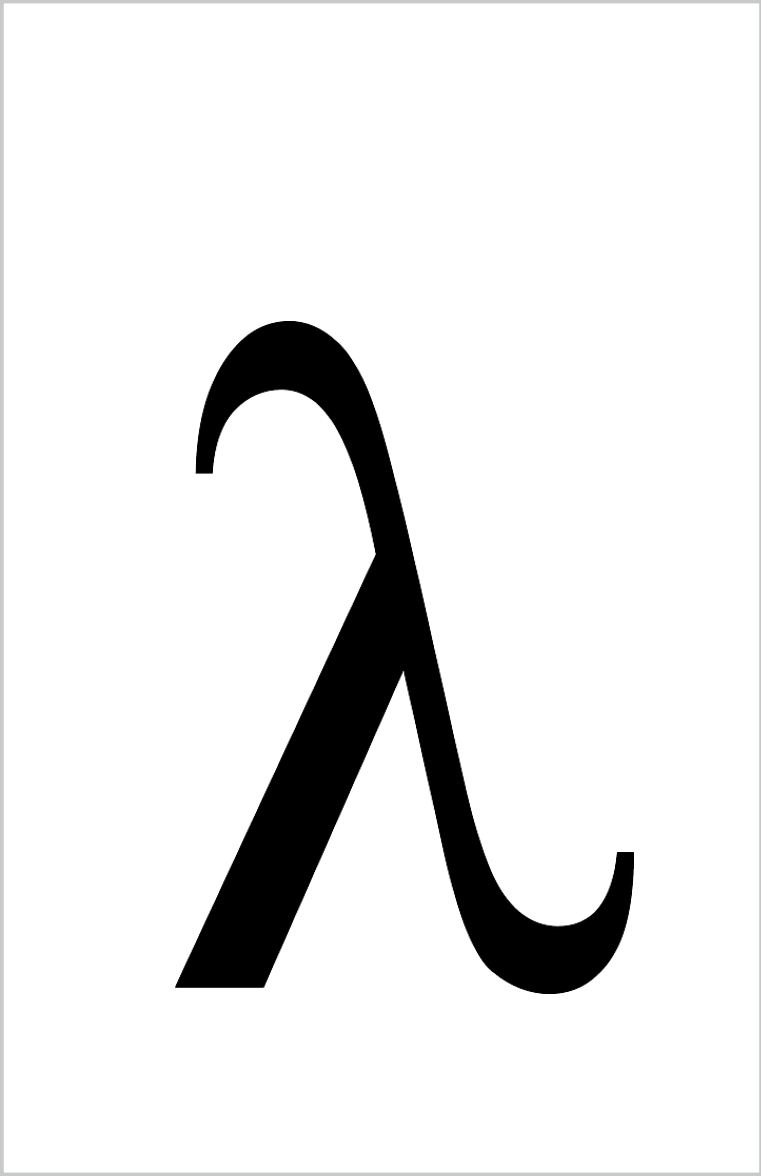



- C#
- Groovy
- Scala
- Javascript
- Go
- Swift
- Rust
- Kotlin
- …

# Scripting languages




- Started with shell languages
- JCL, MS-DOS
- Unix: csh, sed, awk
- Followed by: Perl, Python, Ruby .
- Characteristics
  - Mainly interpreted at runtime
  - Economy of expression
  - No declarations, simple scoping rules
  - Flexible dynamic typing
  - Easy access to system facilities



# Functional programming languages

# The lambda calculus

Alonzo Church
Princeton Prof
1929-1967

- In 1936, Alonzo Church invented the lambda calculus. He called it a logic, but it was a language of pure functions -- the world's first programming language.

"*There may, indeed, be other applications of the system than its use as a logic.*"

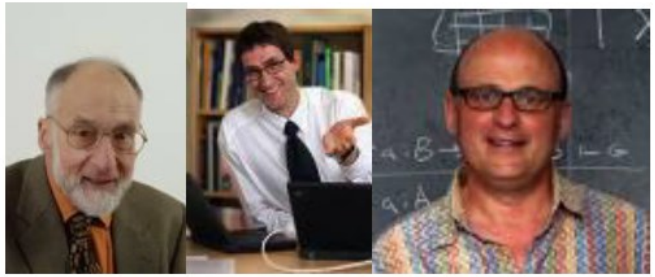
# A bit of history of FP

Alonzo Church:
lambda calculus
1930's

Guy Steele & Gerry Sussman:
Scheme
late 1970's

Xavier Leroy:
Ocaml
1990's

John McCarthy:
LISP
1958

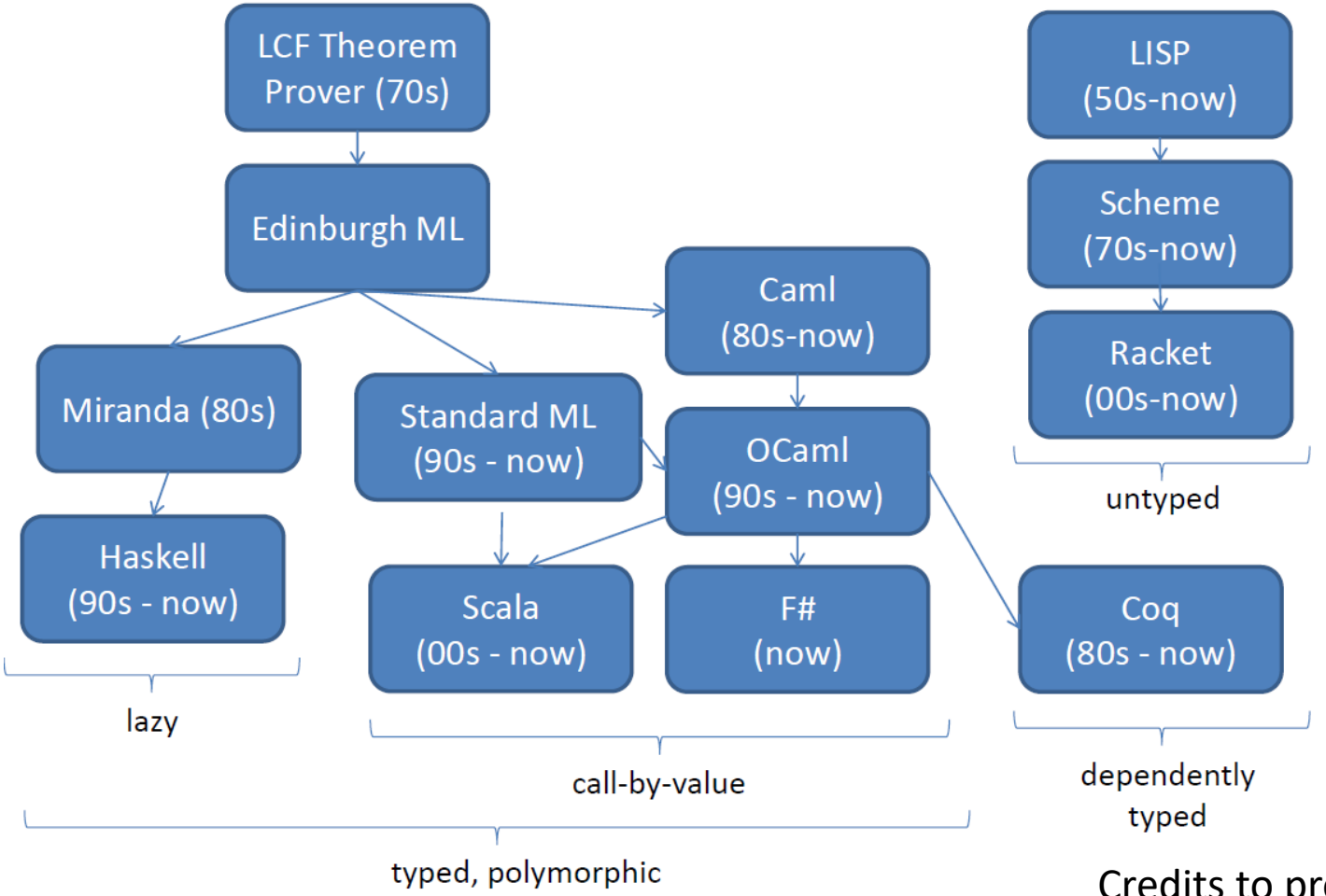


Robin Milner, Mads Tofte, & Robert Harper
Standard ML
1980's

Don Syme:
F#
2000's

[Credits to prof. David Walker]

# Families




Credits to prof. David Walker

# Summary

- What we will see and do in the course
- An overview of the history of the programming languages

# Copyright and credits

- The material is intended solely for students at the University of Trento registered to the relevant course for the Academic Year 2024-2025.
- The material is partially inspired by the material used in previous years, by professor Luca Abeni (Scuola Superiore Sant'Anna), by professor David Walker (Princeton University) and professor Alejandro Serrano (University of Utrecht).

# Next time


- Introduction to ML

Contact me at:

- Chiara Di Francescomarino: c.difrancescomarino@unitn.it