

ST0244 Programming Languages

1. Introduction

Andrés Sicard-Ramírez

Universidad EAFIT

Semester 2023-2

Pedagogical Pact

Course web page

<http://www1.eafit.edu.co/asr/courses/st0244-programming-languages/>

Official channel, exams, programming labs, course's repository, etc.

See course web page.

Pedagogical Pact

Course web page

<http://www1.eafit.edu.co/asr/courses/st0244-programming-languages/>

Official channel, exams, programming labs, course's repository, etc.

See course web page.

Responsibilities

- Lecturer
- Students

Preliminaries

Conventions

- The number and page numbers assigned to chapters, examples, exercises, figures, quotes, sections and theorems on these slides correspond to the numbers assigned in the textbook [Lee 2017].
- The source code examples are in course's repository.

Textbook's First Paragraph

'A career in computer science is a commitment to a lifetime of learning. You will not be taught every detail you will need in your career while you are a student. The goal of a computer science education is to give you the tools you need so you can teach yourself new languages, frameworks, and architectures as they come along.' (p. v)

Initial Remarks

From: 'The Next 7000 Programming Languages' [Chatley, Donaldson and Mycroft 2019].

Initial Remarks

From: 'The Next 7000 Programming Languages' [Chatley, Donaldson and Mycroft 2019].

Evolution:

'Language implementations have evolved to help humans manage this complexity.'
(p. 255)

(continued on next slide)

Initial Remarks

Universal programming language?

'We might hope for a single universal language which is suitable for all niches, as has been a recurring hope since Landin's time. However, the evolutionary model does not predict this. It says nothing about the existence of such a language, and past attempts to create universal languages do not add encouragement.' (p. 279)

(continued on next slide)

Initial Remarks

Which language should I use?

'Another decision point in choosing a language is “get it working” versus “get it right” versus “get it fast/efficient”. In different situations, each might be appropriate, and the software-system context, or niche, determines the fitness of individual languages and hence guides the language choice. A quick script to do some data-processing is obviously quite different from an I/O driver, or the control system of a safety-critical device.' (p. 255)

(continued on next slide)

Popularity of Programming Languages

- TIOBE index: <https://www.tiobe.com/tiobe-index/>
- GitHub: <https://octoverse.github.com/2022/top-programming-languages>

Course Outline

- Introduction
- Syntax
- Object-Oriented Programming
- Functional Programming
- Logic Programming

Programming Paradigms

Definition

A **programming language** is a **formal** language for writing computer programs.

Programming Paradigms

Definition

A **programming language** is a **formal** language for writing computer programs.

Question

What means the 'formal' adjective in the above definition?

Programming Paradigms

Definition

Paradigm:

'A model of something, or a very clear and typical example of something.' (Cambridge Dictionary)

Programming Paradigms

Definition

Paradigm:

'A model of something, or a very clear and typical example of something.' (Cambridge Dictionary)

Definition

Programming paradigms are:

'Ways of thinking about programming.' (p. v)

'High-level approaches for viewing computation.' (Turbark and Gifford 2008, p. 16)

'A way to classify programming languages based on their features.' (Wikipedia, 2019-07-13)

Programming Paradigms

Motivation

A cognitive bias:

'If all you have is a hammer, everything looks like a nail.'

Programming Paradigms

Three programming paradigms

- Imperative/object-oriented programming

E.g. C, C++, COBOL, Fortran, Java, Pascal, Python and Rust.

- Functional programming

E.g. Haskell, Scheme and Standard ML.

- Logic programming

E.g. CLP(R) and Prolog.

Historical Perspective

Remark

The development of programming languages is based in **both** theoretical and engineering developments.

Historical Perspective

Time line*

- c. 1675 Gottfried Wilhelm Leibniz. *Characteristica universalis* (a universal symbolic language). Mechanical calculators.
- 1822 Charles Babbage. Difference engine (mechanical machine for tabulating polynomial functions).
- 1928 David Hilbert and Wilhelm Ackermann. The *Entscheidungsproblem* (decision problem) [Hilbert and Ackermann 1950].
- 1935-6 Alonzo Church. Lambda calculus (computability model) and negative solution to the *Entscheidungsproblem* [Church 1935, 1936].
- 1936-7 Alan Turing. Turing machine (computability model) and negative solution to the *Entscheidungsproblem* [Turing 1936–1937].

(continued on next slide)

*A time line must start in some point and it is necessarily incomplete.

Historical Perspective

Time line (continuation)

- 1939 John Atanasoff and Clifford Berry. The ABC or Atanasoff-Berry Computer. United States.
- c. 1940 Alonzo Church, Alan Turing and Stephen Kleene. The Church-Turing thesis.
- 1943 Tommy Flowers. The Colossus computer. England.
- 1945 John von Neumann. Storing the computer programs (there is controversy about the author(s) of this idea).
- 1946 John Mauchly and J. Presper Eckert. The ENIAC (Electronic Numerical Integrator and Computer). United States.
- 1949 Alan Turing. Design for stored programs and verification of programs [Turing 1949].

(continued on next slide)

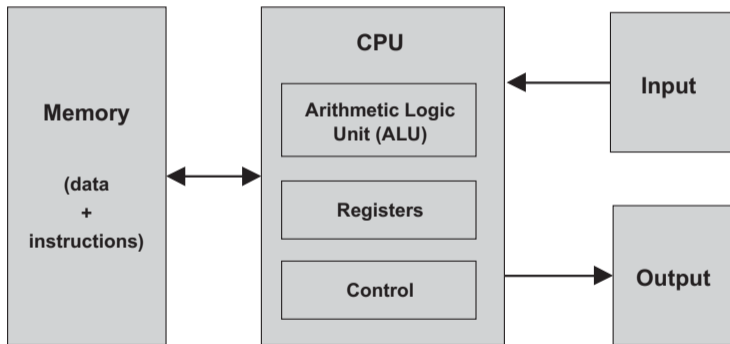
Historical Perspective

Time line (continuation)

- 1957 John Backus and others. **FORTRAN** [Backus, Beeber, Best, Goldberg, Haibt, Herrick, Nelson, Sayre, Sheridan, Stern, Ziller, Hughes and Nutt 1957].
- 1958 John McCarthy. **Lisp** [McCarthy 1960].
- 1960 John Backus and others. **ALGOL 60** [Backus, Bauer, Green, Katz, McCarthy, Perlis, Rutishauser, Samelson, Vauquois, Wegstein, Wijngaarden and Woodger 1960].
- c. 1960 John Backus and Peter Naur. BNF (Backus-Naur Format)
- 1965 J. A. Robinson. The resolution principle [Robinson 1965].
- 1972 Alain Colmerauer and Philippe Roussel. **Prolog**.

Models of Computation

The von Neumann architecture*



*Figure 5.1 in [Nisan and Shimon 2005].

Models of Computation: The Imperative Model

Features

- Decomposition of a program in sub-programs (functions, procedures, sub-routines).
- Structural programming (top-down or bottom-up design).
- Activation records for functions/procedures.
- Division of the data area.

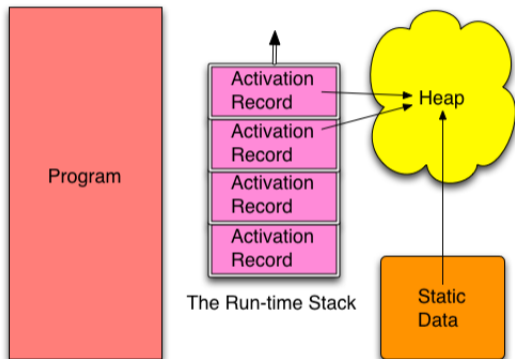


Fig. 1.4

Models of Computation: The Imperative Model

Activation records for each function/procedure invocation

- Local variables.
- The return address (program counter's value before the function/procedure was called).
- Value of parameters.

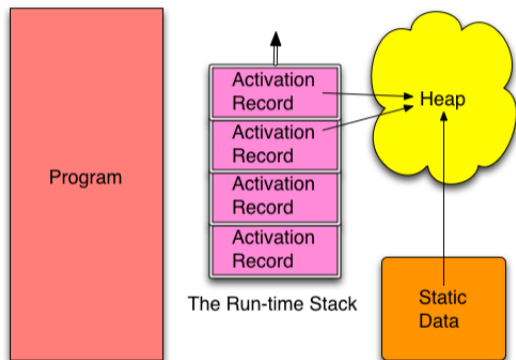


Fig. 1.4

Models of Computation: The Imperative Model

Division of the data area

- Static or global area
Area for storing data and functions that are accessible globally in the program (e.g. constants, global variables, and built-in functions)
- The run-time stack
Area for storing activation records using a LIFO order.
- The heap
Area for dynamic memory allocation (data created at run-time) via references and pointers without pattern to the allocation and deallocation.

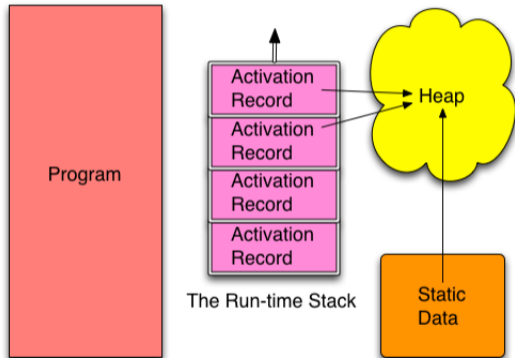


Fig. 1.4

Models of Computation: The Functional Model

Features

- Persistent (immutable) data (cannot be change once created).
- Functions are first-class citizens.
- No difference between program and data.
- Since all the work is made via calling functions the run-time stack is more important than in the imperative model.
- The programmer does not interact with the heap.
- The functional programming is more abstract (good) but the programmer has minor control (bad).

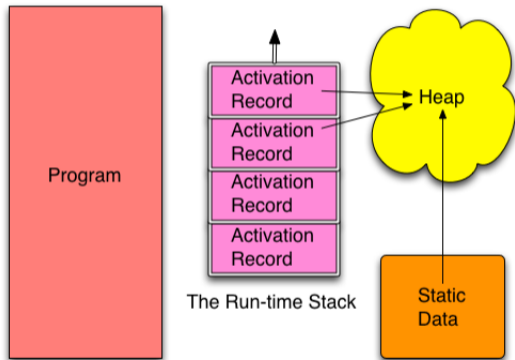


Fig. 1.4

Models of Computation: The Logic Model

Features

- The programmer does not write a program but a database with facts and rules (both are axioms from the logical point of view).
- It is debatable whether we should talk of a division of the data area in the logical model of computation.

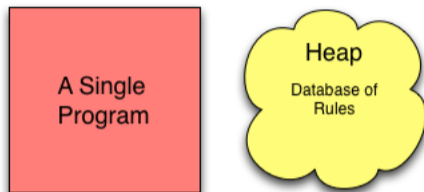


Fig. 1.5

Brief History of Some Programming Languages

Reading

To read the brief history of C, C++, Java, Prolog, Python and Standard ML in the textbook.

Language Implementation

Definition

Machine language is the (binary) language that is read, interpreted and executed by the CPU.

Remark

Machine languages are hardware-dependent.

Language Implementation

Definition

An **assembly language** is a symbolic representation (human readable) of the machine language.

Remark

Assembly languages are hardware-dependent.

Example

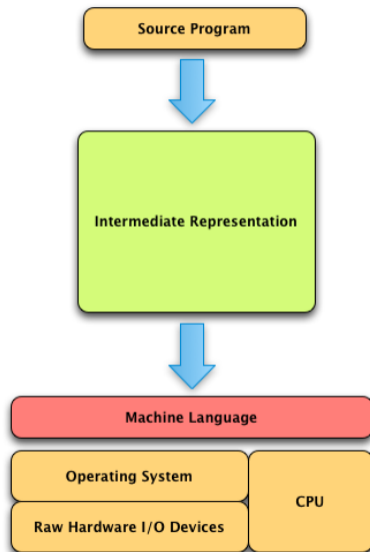
See the `hw/hello-world.asm` file.

Language Implementation

Languages can be implemented in different ways

- A language can be compiled to a machine language.
- A language can be interpreted.
- A language can be implemented by combining compilation and interpretation.

(Fig. 1.11)



Language Implementation

Question

Does the implementation of a programming language depend of the program paradigm represented by the language?

Language Implementation

Question

Does the implementation of a programming language depend of the program paradigm represented by the language? No!

Language Implementation

Question

Does the implementation of a programming language depend of the program paradigm represented by the language? No!

Definition

A **platform** is a specific combination of hardware and operating system.

Language Implementation: Compilation

Definition

A **compiler** is a **program** that converts a source program to machine language.

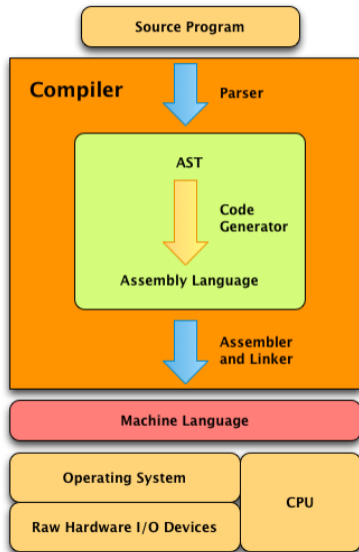
Features

- Abstract syntax tree (AST): Internal representation of the source program.
- If you change your source code you need to recompile.

Remark

Compilers are platform-dependent.

(Fig. 1.12)



Language Implementation: Compilation

Example

C, C++, COBOL, Fortran, Haskell, Pascal and Rust are compiled languages.

Language Implementation: Interpretation

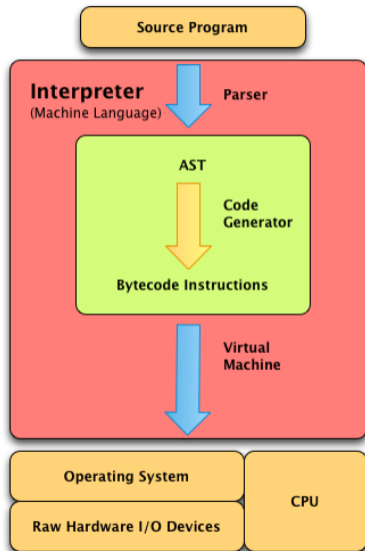
Definition

An **interpreter** is a **program** that executes other programs.

Features

- You execute your source programming by running the interpreter.
- Research problem: Heap memory management.
- Advantage: Portability (the interpreter insulates your program from CPU architecture and operating system dependencies).
- Disadvantage: Speed of execution.

(Fig. 1.13)



Language Implementation: Interpretation

Remark

Interpreters are platform-dependent.

Example

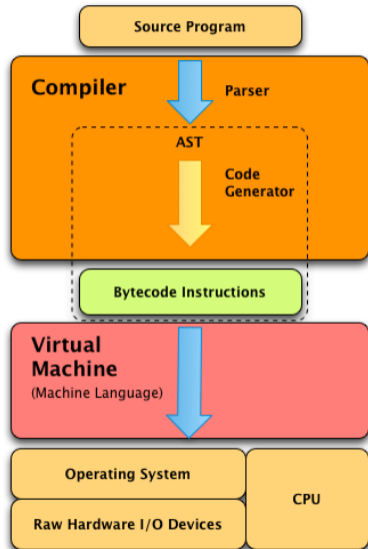
Bash, Haskell, Lisp, Prolog, Python, Ruby and Standard ML are interpreted languages.

Language Implementation: Virtual Machines

Definition

*'A **virtual machine** is a **program** that provides insulation from the actual hardware and operating system of a machine while supplying a consistent implementation of a set of low-level instructions, often called **bytecode**.'* (p. 23)

(Fig. 1.14)

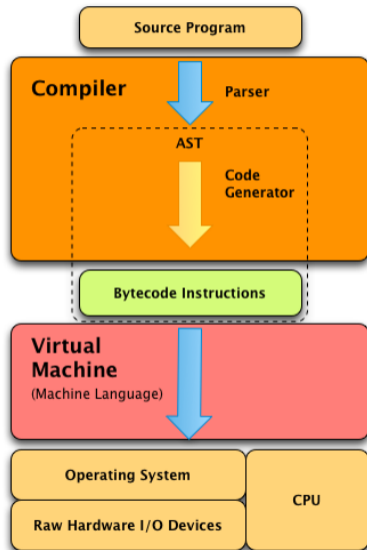


Language Implementation: Virtual Machines

Features

- Separation of the virtual machine from the compiler.
- The programs are compiled to bytecode.
- The bytecode programs are interpreted.
- The interpretation of bytecode programs is faster than the interpretation of source code.
- The programs implemented via virtual machines are more portable than programs implemented via compilers.
- Programs can be distributed in binary (bytecode) form.

(Fig. 1.14)



Language Implementation: Virtual Machines

Remark

Virtual machines are platform-dependent.

Remark

Bytecode instructions are platform-independent.

Example

C#, Java, Python, Standard ML and Visual Basic.Net are implemented via virtual machines.

Types and Type Checking

Types in logic and mathematics

Types as ranges of significance of propositional functions. Let $\varphi(x)$ be a (unary) propositional function. The type of $\varphi(x)$ is the range within which x must lie if $\varphi(x)$ is to be a proposition [Russell 1938, Appendix B: The Doctrine of Types].

In modern terminology, Russell's types are domains of propositional functions.

Example

Let $\varphi(x)$ be the propositional function ' x is a prime number'. Then $\varphi(x)$ is a proposition only when its argument is a natural number.

$$\begin{aligned}\varphi &: \mathbb{N} \rightarrow \{\text{False}, \text{True}\} \\ \varphi(x) &= x \text{ is a prime number.}\end{aligned}$$

Types and Type Checking

Types in programming languages

'They [programming languages] define types to specify which operations make sense on which types of data.' (p. 26)

'A type is an approximation of a dynamic behaviour that can be derived from the form of an expression.' (Kiselyov and Shan 2008, p. 8)

Example

Examples of types include integers, booleans, floating point numbers, characters, strings, lists, Cartesian products (tuples), discriminated unions, sets, functions, recursive/inductive types and user-defined types.

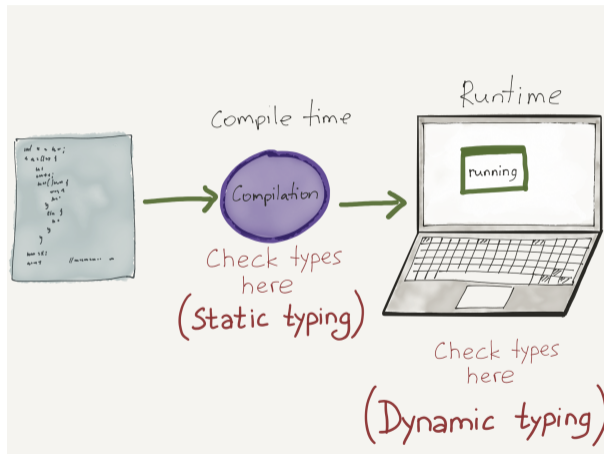
Types and Type Checking

Types systems in programming languages

'A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.' (Pierce 2002, p. 1)

Types and Type Checking

Static typing vs dynamic typing*



*Figure from en.hexlet.io/courses/intro_to_programming/lessons/types/theory_unit.

Types and Type Checking

Example (statically and dynamically typed programming languages)

Dynamically typed: JavaScript, PHP and Python

Statically typed: C, C++, C#, Haskell, Java, Rust and Standard ML

Types and Type Checking*

The static programmer says:

“Static typing catches bugs with the compiler and keeps you out of trouble.”

“Static languages are easier to read because they’re more explicit about what the code does.”

“At least I know that the code compiles.”

“I trust the static typing to make sure my team writes good code.”

“Debugging an unknown object is impossible.”

“Compiler bugs happen at midmorning in my office; runtime bugs happen at midnight for my customers.”

The dynamic programmer says:

“Static typing only catches some bugs, and you can’t trust the compiler to do your testing.”

“Dynamic languages are easier to read because you write less code.”

“Just because the code compiles doesn’t mean it runs.”

“The compiler doesn’t stop you from writing bad code.”

“Debugging overly complex object hierarchies is unbearable.”

“There’s no replacement for testing, and unit tests find more issues than the compiler ever could.”

*From www.smashingmagazine.com/2013/04/introduction-to-programming-type-systems/.

Types and Type Checking

Discussion

'Which is better, dynamically or statically typed languages? It depends on the complexity of the program you are writing and its size. Static typing is certainly desirable if all other things are equal. But static typing typically does increase the work of a programmer up front. On the other hand, static typing is likely to decrease the amount of time you spend testing.' (p. 27)

Types and Type Checking

Definition

Let P be a program.

(i) A type system is **sound** iff

P passed the type checker $\Rightarrow P$ is a correctly typed program.

(ii) A type system is **complete** iff

P is a correctly typed program $\Rightarrow P$ will pass the type checker.

Types and Type Checking

Definition

Let P be a program.

(i) A type system is **sound** iff

P passed the type checker $\Rightarrow P$ is a correctly typed program.





(ii) A type system is **complete** iff

P is a correctly typed program $\Rightarrow P$ will pass the type checker.








Example

The **Standard ML** type system is sound and complete.






References

-  Backus, J. W., Bauer, F. L., Green, J., Katz, C., McCarthy, J., Perlis, A. J., Rutishauser, H., Samelson, K., Vauquois, B., Wegstein, J. H., Wijngaarden, A. van and Woodger, M. (1960). Report on the Algorithmic Language ALGOL 60. Communications of the ACM 3.5. Ed. by Naur, Peter, pp. 299–314. DOI: [10.1145/367236.367262](https://doi.org/10.1145/367236.367262) (cit. on p. 21).
-  Backus, J. W., Beeber, R. J., Best, S., Goldberg, R., Haibt, L. M., Herrick, H. L., Nelson, R. A., Sayre, D., Sheridan, P. B., Stern, H., Ziller, I., Hughes, R. A. and Nutt, R. (1957). The FORTRAN Automatic Coding System. In: Proceedings Western Joint Computer Conference, pp. 188–198 (cit. on p. 21).
-  Chatley, Robert, Donaldson, Alastair and Mycroft, Alan (2019). The Next 7000 Programming Languages. In: Computing and Software Science. State of the Art and Perspectives. Ed. by Steffen, Bernhard and Woeginger, Gerhard. Vol. 10000. Lecture Notes in Computer Science. Springer, pp. 250–282. DOI: [10.1007/978-3-319-91908-9_15](https://doi.org/10.1007/978-3-319-91908-9_15) (cit. on pp. 6, 7).
-  Church, Alonzo (1935). An Unsolvable Problem of Elementary Number Theory. Preliminar Report (Abstract). Bulletin of the American Mathematical Society 41.5, pp. 332–333. DOI: [10.1090/S0002-9904-1935-06102-6](https://doi.org/10.1090/S0002-9904-1935-06102-6) (cit. on p. 19).

References

-  Church, Alonzo (1936). An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics* 58.2, pp. 345–363. DOI: [10.2307/2371045](https://doi.org/10.2307/2371045) (cit. on p. 19).
-  Hilbert, D. and Ackermann, W. [1938] (1950). *Principles of Mathematical Logic*. 2nd ed. Translation of the second edition of *Grundzüge der Theoretischen Logik*, Springer, 1938. Translated by Lewis M. Hammond, George G. Leckie and F. Steinhardt. Edited and with notes by Robert E. Luce. Chelsea Publishing Company (cit. on p. 19).
-  Kiselyov, Oleg and Shan, Chung-chieh (2008). Interpreting Types as Abstract Values. *Formosan Summer School on Logic, Language and Computacion (FLOLAC 2008)* (cit. on p. 43).
-  Lee, Kent D. [2014] (2017). *Foundations of Programming Languages*. 2nd ed. Undergraduate Topics in Computer Science. Springer (cit. on p. 4).
-  McCarthy, John (1960). Recursive Functions of Symbolic Expressions and their Computation by Machine, Part I. *Communications of the ACM* 3.4, pp. 184–195. DOI: [10.1145/367177.367199](https://doi.org/10.1145/367177.367199) (cit. on p. 21).
-  Nisan, Noam and Shimon, Schocken (2005). *The Elements of Computing Systems. Building a Modern Computer from First Principles*. MIT Press (cit. on p. 22).
-  Pierce, Benjamin C. (2002). *Types and Programming Languages*. MIT Press (cit. on p. 44).

References

-  Robinson, J. A. (1965). A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM* 12.1, pp. 23–41. DOI: [10.1145/321250.321253](https://doi.org/10.1145/321250.321253) (cit. on p. 21).
-  Russell, Bertrand [1903] (1938). *The Principles of Mathematics*. 2nd ed. W. W. Norton & Company, Inc (cit. on p. 42).
-  Turbark, Franklyn and Gifford, David (2008). *Design Concepts in Programming Languages*. MIT Press (cit. on pp. 14, 15).
-  Turing, Alan M. (1936–1937). On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceeding of the London Mathematical Society* s2-42, pp. 230–265. DOI: [10.1112/plms/s2-42.1.230](https://doi.org/10.1112/plms/s2-42.1.230) (cit. on p. 19).
-  — (1949). Checking a Large Routine. In: *Report of a Conference on High Speed Automatic Calculating* (cit. on p. 20).