1 Introduction

"Principles of programming languages"

Mark Armstrong, PhD Candidate, McMaster University

Fall, 2019

1 Preamble

1.1 Notable references

- Concepts of Programming Languages, Sebesta, 10th ed.
  - Section 1.3 - Language Evaluation Criteria
  - Section 5.4 - The Concept of Binding
  - Section 5.5 - Scope
- Concepts, Techniques, and Models of Computer Programming
  - Preface
  - Section 2.1.1 - Language syntax

1.2 Update history

Sept. 12
- The description of the lifetime of stack dynamic variables has been changed.
  - From: "lifetime is the amount of runtime spent in that scope or sub-scopes".
  - To: "lifetime is the portion of runtime between when the unit of code containing the variable is entered and when control returns to the invoker of the unit of code."

1.3 Table of contents

- Welcome!
  - Instructor: Mark Armstrong
  - Teaching assistant: Musa Alhassy
  - Teaching assistant: Maryam Kord
- About the notes and slides for this course
  - Emacs
  - Org mode
  - Literate programming
- Purpose and goals of this course
  - Informal objectives
  - Course preconditions
  - Course postconditions
  - Formal rubric for the course
- Principles of programming languages
  - A brief definition of "programming language"
  - The two components of every programming language
    * Implementation strategy: another component?
  - Paradigms: classifying programming languages
    * The commonly discussed paradigms
    * A taxonomy of programming paradigms
    * One language, many paradigms
    * Exercise: On paradigms
  - Subjective criteria for comparing and evaluating languages
    * Readability
    * Writability
    * Reliability
    * Cost
- Key concepts and terminology
  - Compiled, interpreted, and hybrid approaches
    * Compilation
    * Interpreters
    * Hybrid methods
    * Exercise: Researching implementations
  - Abstraction
    * Levels of abstraction
  - Static vs. dynamic
  - Scope

    - Exercise: On dynamic scoping
    - Exercise: Entities which introduce scope
  - Variables and memory
    - The data segment, stack, and heap
    - Kinds of memory allocation
    - Exercise: Memory allocation in C++
    - Lifetime
    - Garbage collection
- Exercises

2 Welcome!

Welcome to the course!

2.1 Instructor: Mark Armstrong

- Email: armstmp “at” mcmaster.ca
- Website: http://www.cas.mcmaster.ca/~armstmp/
- Office: ITB-229

2.2 Teaching assistant: Musa Alhassy

- Email: alhassm "at" mcmaster.ca
- Website: `https://alhassy.github.io`
- Office: ITB-229

2.3 Teaching assistant: Maryam Kord

- Email: kordm "at" mcmaster.ca

3 About the notes and slides for this course

- The notes/slides for this course are developed in Emacs using Org mode.
  - Along with almost everything else for the course!
- Specifically,
  - the slides are created by exporting the Org files to HTML using the org-reveal package, and
  - the document style `pdf`'s are created by exporting the Org files to LaTeX.

- The slides are laid out in two dimensions.
  - When viewing them in your browser, press `?` to see a help page describing the possible interactions.
  - Most importantly, press `esc` or `o` to see an overview of the slide deck.

3.1 Emacs

Emacs is

> An extensible, customizable, free/libre text editor — and more.
>
> At its core is an interpreter for Emacs Lisp, a dialect of the Lisp programming language with extensions to support text editing.
>
> — GNU Emacs homepage

- The learning curve is fairly steep compared to most similar tools, but:
  - Distributions (configurations) such as Spacemacs and doom-emacs optionally ease the learning curve.
  - As the longest lived extensible text editor, Emacs has an unrivaled ecosystem of extensions.
  - Emacs is best configured *by writing code*, a natural exercise of your talents.
- Especially if you have not settled on an editor for general use, check out Emacs!

3.2 Org mode

Even if you do not use Emacs, I recommend checking out Org mode.

> Org mode is for keeping notes, maintaining TODO lists, planning projects, and authoring documents with a fast and effective plain-text system.
>
> — Org mode homepage

While Org mode is used in plain-text documents, it supports exporting to various formats, including `pdf`'s (through LaTeX), `html`, and `odt`, all “out of the box”!

- Write once, export many.

Packages for Org mode are available for both VSCode and Atom.

3.3 Literate programming

Org mode is especially useful in that it provides support for *literate programming.*

> Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.
>
> — Donald E. Knuth, "Literate Programming"

Regardless of the tools you choose to use, in this course it will be expected that you follow this philosphy.

- The purpose of your programs here is to *explain to the course staff what you want the computer to do.*

That is, documentation is not optional, and not to be an afterthought.

4 Purpose and goals of this course

The undergraduate calendar description of this course is

> Design space of programming languages; abstraction and modularization concepts and mechanisms; programming in non-procedural (functional and logic) paradigms; introduction to programming language semantics.

But we should be more specific.

4.1 Informal objectives

- Expose you to several programming languages.
  - A relatively shallow but comprehensive survey.
  - Focussing on general-purpose languages.
- *Formally* describe programming language syntax and semantics.
  - An application of theory you have learned previously.
- Analyse the building blocks of languages.

  - Programs can be *massive* entities, but they are composed of relatively few *small* pieces.

- Provide you with criteria by which to judge languages.

  - So you can identify what languages fit what tasks.

- Examine the origins of certain languages/groups of languages.

  - Historical context provides insight into why languages are designed the way they are.

4.2 Course preconditions

Before beginning this course:

1. Students should know and understand:

   (a) Basic concepts about integers, sets, functions, & relations.
   (b) Induction and recursion.
   (c) First order logic, axiomatic theories & simple proof techniques.
   (d) Regular expressions & context-free grammars.
   (e) Programming in imperative language
   (f) Basic concepts of functional programming languages.

2. Students should be able to:

   (a) Produce proofs involving quantifiers and/or induction.
   (b) Understand the meaning of a given axiomatic theory.
   (c) Construct regular sets & context-free languages.
   (d) Produce small to medium scale programs in imperative languages.
   (e) Produce small scale programs in functional languages.

4.3 Course postconditions

After completion of this course:

1. Students should know and understand:

   (a) The basics of several programming languages.

(b) Formal definitions of syntax & semantics for various simple programming languages.

(c) Various abstraction & modularisation techniques employed in programming languages.

2. Students should be able to:

   (a) Reason about the design space of programming languages, in particular tradeoffs & design issues.

   (b) Produce formal descriptions of syntax & semantics from informal descriptions, identifying ambiguities.

   (c) Select appropriate abstraction & modularisation techniques for a given problem.

   (d) Produce (relatively simple) programs in various languages, including languages from non-procedural paradigms.

4.4 Formal rubric for the course

Topic	Below	Marginal	Meets	Exceeds
Familiarity with various programming languages (PLs)	Shows some competence in procedural languages, but not languages from other paradigms	Shows competence in procedural languages and limited competence in languages from other paradigms	Achieves competence with the basic usage of various languages	Achieves competence with intermediate usage of various languages
Ability to identify and make use of abstraction, modularisation constructs	Cannot consistently identify such constructs	Identifies such constructs, but does not consistently make use of them when programming	Identifies such constructs and shows some ability to make use of them when programming	Identifies sucj constructs and shows mastery of them when programming
Ability to comprehend and produce formal descriptions of PL syntax	Unable or rarely able to comprehend given grammars; does not identify ambiguity or precedence rules	Comprehends given grammars, but produces grammars which are ambiguous or which do not correctly specify precedence	Makes only minor errors regarding precedence or ambiguity when reading or producing grammars	Consistently fully understands given grammars and produces correct grammars.
Ability to comprehend and produce operational semantics for simple PLs	Rarely or never comprehends such semantic descriptions	Usually comprehends such semantic descriptions, but cannot consistently produce them	Comprehends such semantic descriptions and produces them with only minor errors	Comprehends such semantic descriptions and produces them without errors
Ability to comprehend denotational and axiomatic semantics for simple PLs	Rarely or never comprehends such semantic descriptions	Inconsistently comprehends such semantic descriptions	Consistently comprehends such semantic descriptions	Consistently comprehends and can produce some simple semantic descriptions

5 Principles of programming languages

Before we begin in earnest, let us set the stage by answering some basic questions.

- What are “programming languages”?
- What are their components?
- How can we compare, group or classify them?

5.1 A brief definition of “programming language”

- A finite, formal language for describing (potentially) infinite processes.
  - Usually the language is actually quite small!
    - A CFG for *all of C* has only 91 nonterminals (as constructed in the C11 international standard, appendix 2).
    - We will construct simple languages with far less than 100 *productions*.
  - *Formality* is required because we want our programs to be run on machines.
    - Machines fair poorly with informal language.
  - Considering just the set of commands executable by the processor, modern computers are fairly simple machines.

5.2 The two components of every programming language

Languages can be distinguished by two components:

- **Syntax**: the rules for the form of legal programs.
- **Semantics**: the rules for the meaning of legal programs.

> *Colourless green ideas sleep furiously.*

Change either, and you have a different language.

5.2.1 Implementation strategy: another component?

Sometimes *implementation strategy* is also considered a component of a programming language.

- i.e., whether it is compiled or interpreted.

We do not take this view.

- Implementation strategy is external to the language.
- A language may have many compilers, interpreters, runtime environments, etc.

5.3 Paradigms: classifying programming languages

Classifications of programming languages based on what kind of abstractions they (natively) support.

- One language may support many paradigms.
- It is often possible to write programs following a paradigm the language does not natively support.
  - For instance, following object oriented principles when writing C code.
  - Native support is important!
    * It frees the programmer from having to "hold" the abstraction in their own mind.
    * We will see time and time again that well designed languages save the programmer work and mental effort!
- Supported abstractions should be *orthogonal*; separate from each other, but able to be combined in sensible, predictable ways.
  - Ideally there is a hierarchy of abstractions.

5.3.1 The commonly discussed paradigms

It is common to consider only a small number of paradigms, usually in this hierachy.

Imperative/Procedural		Declarative	
Object-oriented	Non-Object Oriented	Functional	Logic

But this is far from a complete picture!

5.3.2 A taxonomy of programming paradigms

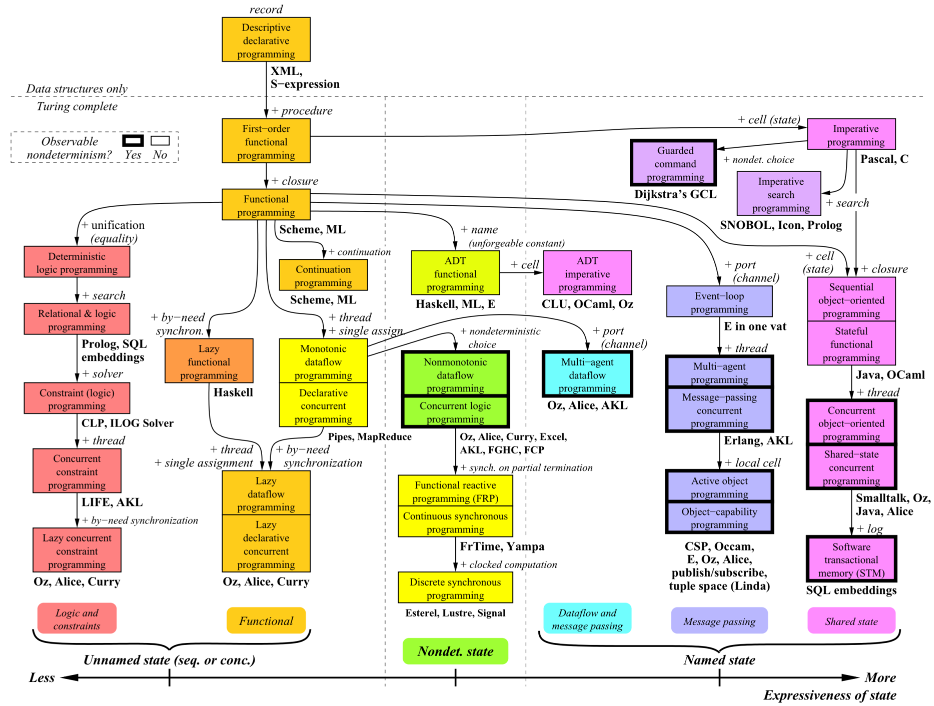

Figure 2. Taxonomy of programming paradigms

> — Van Roy, P. "Programming Paradigms for Dummies: What Every Programmer Should Know"

5.3.3 One language, many paradigms

Different paradigms lend themselves to different tasks, so it is beneficial for a language to support more than one.

But then, "More is not better (or worse), just different"!

> All computation models have their place. It is not true that models with more concepts are better or worse. This is because a new concept is like a two-edged sword. Adding a concept to a computation model introduces new forms of expression, making some programs simpler, but it also makes reasoning about programs harder.

For example, adding mutable variables to functional languages enables object-oriented programming techniques.

But reasoning in the presence of mutable variables is notoriously difficult!

5.3.4 Exercise: On paradigms

Pick a language you are familiar with, and research what paradigms it supports. (Don't expect to find a definite yes/no answer for every paradigm from Van Roy's taxonomy).

Then try to determine paradigms which are not natively supported, but are still feasible to follow when coding.

5.4 Subjective criteria for comparing and evaluating languages

We can use a mixture of *objective* and *subjective* criteria to compare and evaluate languages we discuss.

We use four main subjective criteria. Note there is overlap!

- Readability
- Writability
- Reliability
- Cost

5.4.1 Readability

- How easily can code be interpreted by humans?
  - How much "noise" is there?
- How easily can errors be detected by visual inspection?
- How many constructs must the reader be familiar with?
- How many ways can constructs be combined?
  - Do the combinations behave in predicatable ways?

The last two points deal with *orthogonality*.

> Orthogonality in a programming language means that a relatively small set of primitive constructs can be combined in a relatively small number of ways to build the control and data structures of the language.
>
> — Concepts of Programming Languages 10th ed.

5.4.2 Writability

- How easily can humans create code?
- How many abstractions are supported?
  - How easily can abstract ideas be captured in the language?
  - Of course this depends upon how well the idea fits the paradigm!
- How much *syntactic sugar* is available?
  - Syntactic sugar is syntax which does not add constructs/features, only (presumably simpler) ways to express existing constructs/features.

> For instance, the actor model of concurrency naturally fits in an object oriented language, but may not fit as well in a functional language.
>
> But the functional reactive model of concurrency will fit better in the functional paradigm.

5.4.3 Reliability

- To what extent is *syntactically valid* code guaranteed to *be* valid?
  - What types of errors are caught before runtime?
- What tools are provided to the programmer for recovering from errors that do occur?
- Are "unsafe" features included in the language?
- What tools are provided/available to the programmer for *specifying* and *verifying* code?
  - Formal specification involves using mathematical language to describe the meaning of programs.

        * *Axiomatic semantics.*
        * Sometimes, automated tools can check code meets formal specifications.
        * See CS3EA3: Software Specifications and Correctness.

5.4.4 Cost

- How much work is involved in
  - Training programmers in a language?
  - Writing new code?
    * Writability!
  - Maintaining a code base?
    * Readability and reliability!
- Are good programming tools/environments available? Do they have a cost?
- How efficient are compilers for the language (if code is to be compiled)?
- How efficient is compiled code (or the interpreter, if the code is to be interpreted).
  - What optimisation options are availble for compiled code?

> There is a tradeoff between efficiency of the compiler and the amount it optimises code.
>
> - For ease of development, you want quick compilation, and can sacrifice efficiency of the compiled code.
> - For a good end product, you want efficient compiled code, even if compilation takes longer.

6 Key concepts and terminology

Following are some key concepts and terms which

- we will either see repeatedly throughout the course, or
- which we should clarify before we move on.

(Some of this information simply won't have a better place to go).

6.1 Compiled, interpreted, and hybrid approaches

Languages exist in a hierarchy.

- We are focused on the "upper portion" of the hierarchy.
  - High-level languages!
  - We may use the term low-level for some languages, C in particular, but this is relative.
    - C is much higher level than assembly!
- We need to translate into machine code to run our programs.
  - Compilation, interpretation and "hybrid approaches" are strategies for doing this.

6.1.1 Compilation

Translate the whole program (and any libraries or other code resources needed) ahead of running it.

- High upfront cost (time), for increased efficiency at runtime
- Not portable; machine code is machine dependent.

6.1.2 Interpreters

Translate the program *as we are running it.*

- No upfront cost, but less efficient.
- Portable; can be run on any machine with an interpreter.
  - Alleviates some of the programmer's responsibility.
    - One user (or group) writes the interpreter *once* (per machine type); it can be used by any number of users for any number programs.
- Efficiency is improved by using **just-in-time compilation**.
  - Store the result of interpretation so it can be used again.
- Can achieve better error reporting.
  - Relationship between original and translated codes is known at runtime.
  - This relationship is discarded when compiling code.

6.1.3 Hybrid methods

Compile into an intermediate language, which is then translated into assembly.

- This intermediate language is usually similar to assembly.
  - But targets a virtual machine, not actual hardware!
- Usually called *bytecode*.
- Greatly offsets efficiency cost of interpretation.
- More portable than compiled code; just need a bytecode interpreter for each target machine.

6.1.4 Exercise: Researching implementations

Research for yourself what implementations are available for languages you are familiar with.

Consider:

- Can you find different approaches (compiler, interpreter, hybrid) for the language?
- Is one approach "the norm" or "preferred" for the language?
- What is the availability for each implementation?
  - Who maintains them?
  - Are they open source?

6.2 Abstraction

The concept of an *abstraction* is central to programming languages.

> We define an abstraction loosely as a tool or device that solves a particular problem.
>
> — Concepts, Techniques, and Models of Computer Programming

> Briefly, abstraction means the ability to define and then use complicated structures or operations in ways that allow many of the details to be ignored.
>
> — Concepts of Programming Languages 10th ed.

6.2.1 Levels of abstraction

Highly abstract (User knows fewer details)
A writing instrument ↓ A pencil ↓ A mechanical pencil ↓ This pencil
Highly concrete (User knows all details)

Language `A` is "higher-level" than language `B` if it allows you to work at a higher level of abstraction than language `B`.

> For instance, C++ and C# are higher-level than C, because as a modularisation construct, classes hide more implementation details than the C alternative of defining structs and procedures using them.
> (Pure) functional languages are higher-level than imperative languages, because the user need not be aware of the state of memory.
>
> Logic lanauges are even higher-level than functional languages, because the programmer does not concern themselves with the algorithm, only the problem description.

6.3 Static vs. dynamic

We will see these terms in particular applied to many concepts.

- Static: set before runtime; fixed throughout the runtime.
  - Sometimes the phrase "compile time" is used.
  - But "static" may also mean "load time".
    - I.e., the time at which the program is loaded into memory.
  - And if we are using an interpreter, there is no "compile time"!
- Dynamic: set during runtime; (potentially) changes throughout runtime.

6.4 Scope

The *scope of an entity* is the portion of the program in which it is “visible”.

- Usually scope is statically determined.
  - So the scope of entities can be determined by reading the code.
  - With dynamic scoping, the scope of entities depends upon the “execution trace”, i.e., the “path” through the program.
  - Dynamic scoping is definitely the exception.
    - The detriments strongly outweigh the benefits.

A *scope* is a portion of the program to which the visibility of entities may be limited.

- Design decision: what constructs introduce scopes?
  - Almost certainly subroutines do.
  - Do conditional branches? Loop bodies?

6.4.1 Exercise: On dynamic scoping

- Historically, Lisps (languages directly descended from Lisp) were dynamically scoped.
- Presently, the only widely used Lisp which uses dynamic scoping by default is Emacs Lisp (eLisp).

Try out this snippet of Emacs lisp code, either in Emacs or using an online REPL.

```
(let ((x 2) (y 5)) ; "global" variables x and y
  (progn
    (defun mult-x-y ()
      (* x y)) ; returns x * y

    (defun A ()
      (let ((x 3)) ; local variable x
        (mult-x-y)))

    (defun B ()
      (let ((y 10)) ; local variable y
```

```
      (A)))

  (message "mult-x-y returns %d, A returns %d and B returns %d"
    (mult-x-y) (A) (B))))
```

1. Ensure you understand the results.
2. Using this understanding, formulate some advantages and disadvantages of dynamic scoping.

6.4.2 Exercise: Entities which introduce scope

In a few different programming languages, investigate whether condition branches and loop bodies introduce scopes.

I.e., test out code such as

```
if B then
  int x = 0
endif

y = x   % Is x still in scope?
```

Specifically, investigate languages which have *iterating loops*, (usually called `for` loops). What is the scope of an iterator for such loops?

6.5 Variables and memory

- A *variable* is an abstraction of a memory cell.
  - Come in many forms.
  - Typically we use *variable* to mean a named, mutable cell of memory.
  - When possible, we use more specific words.
    - Argument, parameter, constant, reference, etc.
  - Or qualify the word variable.

6.5.1 The data segment, stack, and heap

In the context of running a program (or a thread of a program):

- The *data segment* is a portion of memory used to store the program itself and static/global variables.

- The *stack* is an organised portion of memory allocated for the program.
  - Blocks at the "top" of the stack are reserved when entering units of code.
    - Units of code include subroutines and/or unnamed blocks, depending upon the language.
  - They are *popped off* the stack when leaving the unit of code.
- The *heap* is an unorganised portion of memory allocated for the program.
  - Allocation/deallocation may be independent of units of code.
  - Disorganisation can lead to inefficiency and other problems.
    - Cache misses, page faults, etc.
    - Running out of memory or appropriately sized continguous blocks.
    - Garbage, dangling references, etc.

6.5.2 Kinds of memory allocation

We can distinguish four kinds of memory allocation.

- Static
  - Allocation is done *before runtime* (static), in the data segment.
    - At load time, when the program is loaded into memory. (Not compile time).
- Stack dynamic
  - Allocation is dynamic, automatic, and on the stack.
  - Amount of memory needed must be known when entering their scope.
  - This is "the usual method" in most imperative languages.
- Implicit heap dynamic
  - Allocation is dynamic, automatic, and on the heap.
  - Memory is simply allocated when the variable is assigned; the programmer does not need to "do anything".

- Explicit heap dynamic
  - Allocation is dynamic, *manual*, and on the heap.
  - The programmer must use a command to allocate memory *before* assignments can be made.
    * E.g., `malloc` or `new`.
  - In fact these blocks of memory can be considered *nameless variables*; we need other, *reference* variables to *point* to their address.

6.5.3 Exercise: Memory allocation in C++

Investigate:

- Which of the five kinds of memory allocation are available in C++.
- How each kind of memory allocation is denoted.

6.5.4 Lifetime

The *lifetime* of a variable is the portion of the runtime (not the code) during which it has memory allocated for it.

- There may be several *instances* of a single variable in the code, each with its own lifetime.
- The lifetime of a variable depends upon the memory allocation strategy used for it.
  - Static: lifetime is the whole of runtime.
  - Stack dynamic: lifetime is the portion of runtime between when the unit of code containing the variable is entered and when control returns to the invoker of the unit of code.
    * In the case of functions/procedures, local variables end their lifetime when the function/procedure returns.
    * Note that a variable go out of scope, but still be alive.
  - Implicit heap-dynamic: lifetime begins when assigned. The end of lifetime depends upon garbage collection.
  - Explicit heap-dynamic: lifetime begins when the allocation command is used. The end of lifetime may be
    * when the programmer issues a deallocation command, or
    * depend upon garbage collection.

6.5.5 Garbage collection

Definition 6.1. *Garbage*: allocated memory which is no longer accessible.

Definition 6.2. *Garbage collection*: deallocating garbage.

- A complex activity.
  - We will discuss the basics here, and then likely not touch on it again.
  - Has a large influence on the efficiency of compiled/interpreted code.
- Two main categories of garbage collection algorithms.
  - Tracing
    - Which includes the "mark & sweep" algorithm.
  - Reference counting
- Additionally, language implementations may include compile time "escape analysis".
  - Memory may be allocated off the stack instead of the heap *if* no references to the variable are available outside its scope (if it doesn't "escape").
- Tracing
  - Usually triggered when a certain memory threshold is reached.
  - Starting from some set of base references (usually those in the stack and data segment), "trace" references, somehow marking those that are reachable.
    - Starting from the set of base references, "determine the transitive closure of reachability".
  - Once done, free memory that is not marked.
  - "Naive" mark & sweep:
    - Each object in memory has a bitflag for this "marking", kept clear except during garbage collection.
    - When activated, traverse all reachable memory, "marking" what's reached, then "sweep" away all unmarked memory (and reset the flags of marked memory).

    - Have to pause the program to do this.
    - Better methods do not require pausing.

- Reference counting
  - An "eager" approach.
  - As the program runs, keep a count of the number of references to every object in memory.
    - Must be constantly monitoring!
  - When the number of references reaches zero, can free the memory for that object.
  - Have to account for cycles!
    - Which increases the overhead.

7 Exercises

- On paradigms
- Researching implementations
- On dynamic scoping
- Entities which introduce scope
- Memory allocation in C++