⛳

Chapter 3: Mr Jamilu continuation

Type	Lecture
Materials	Chapter 3.pdf
Reviewed	☐

💡 **Notion Tip:** Add more details right in this page, including toggles, images, and webpage links, and more. Break up the page by using our different header options so that it's easier to read. Learn more about different types of content blocks here.

Course Goals and Objectives

- **Learn the mechanics of programming languages.**
- **Explore diverse programming languages**, recognizing that each language has strengths and weaknesses suited for specific tasks, incorporates different paradigms, and adapts to advancements in hardware.
- **Recognize valuable programming patterns** and comprehend how languages are formally defined and implemented.
- **Learn fundamental programming language concepts.**
- **Improve programming skills** by practicing new languages and programming styles.

Course Structure

The course will cover:

- Dynamic/Scripting Languages (Ruby)
- Functional Programming (OCaml)
- Scoping, type systems, parameter passing
- Regular expressions & finite automata
- Context-free grammars & parsing
- Lambda Calculus
- Logic programming (Prolog)
- Secure programming
- Comparing language styles; other topics

The course will incorporate a variety of teaching and assessment methods:

- **Lectures**: introduce the course content.
- **Discussion Sections**: deepen understanding through interactive activities and programming exercises.
- **Clicker Quizzes**: in-class quizzes combined with lectures.
- **Projects**: 6 projects using Ruby, OCaml, Prolog, and security concepts.
- **Tests**: 5 quizzes (lowest dropped), 2 midterms, 1 final exam.

Course Policies

- **Required Technology**: Turning Technology clicker or Phone App (free subscription).
- **Laptop Use**: Laptops and tablets are not allowed in class except for note-taking in the back of the class. Cell phones must be kept quiet.
- **Project Grading**: Projects will be graded using the submit server on the Grace cluster. Students are responsible for ensuring their programs run correctly on the cluster, though they can develop programs on their own machines.
- **Learning Resources**: Lecture notes are the primary text, supplemented with readings and the internet. Students are accountable for all material in the notes.

- **Academic Integrity**: All written work must be original. Collaboration is allowed on high-level project questions and practice exam questions, but students must not copy code or share their code with others.

Key Concepts in Programming Languages

Turing Completeness

Almost all general-purpose programming languages are Turing complete, meaning they can compute any function computable by a Turing Machine. While any program can theoretically be written in any Turing complete language, different languages are better suited for different tasks and applications.

Benefits of Studying Programming Languages

- **Improved programming skills:** Programming is a human activity, and understanding how language features impact programming can help you write better code for specific applications.
- **Easier adaptation to new languages:** Learning different paradigms and language design concepts can make it easier to pick up new languages and apply ideas across languages.

Evolution of Programming Language Goals

- **1950s-60s**: Focused on efficient program execution and hardware-centric language features due to expensive machines and relatively inexpensive programmers.
- **Present**: Focus has shifted to design concepts, programmer productivity, and managing resources like communication, effort, power, privacy, etc. due to inexpensive machines and relatively expensive programmers.

Language Attributes

- **Syntax**: The rules governing how a program is written. While syntax can vary between languages, these differences are usually superficial and easy to overcome.

- **Semantics**: The meaning of a program and what it does. Semantics can be specified informally or formally (mathematically), with formal semantics providing a precise and unambiguous definition of program behavior.
- **Paradigm**: A way of thinking about and expressing computation. Different paradigms favor different approaches, and understanding paradigms can help you choose the right tool for the job. This course covers imperative, logic, functional, and scripting/dynamic paradigms.
- **Implementation**: How a program is executed. Languages can be implemented through compilation or interpretation.

Programming Language Paradigms

- **Imperative**: Also called procedural or von Neumann, this paradigm focuses on procedures and statements, with programs typically modifying memory. Examples include FORTRAN, Pascal, and C.
- **Functional**: Emphasizes immutability, where variables are never re-defined. Functions are treated as first-class citizens and can be passed as arguments and returned as results. Examples include LISP, ML, Scheme, Haskell, and OCaml.
- **Logic**: Also known as rule-based or constraint-based, this paradigm defines program rules that constrain possible outcomes. Evaluation involves constraint satisfaction through search. Examples include PROLOG and Datalog.
- **Object-Oriented**: Centers around objects that encapsulate data and functions, often organized into classes with inheritance. The underlying basis can be imperative or functional. Examples include Smalltalk, C++, OCaml, Ruby, and Java.
- **Dynamic (Scripting)**: Often used for rapid prototyping and tasks like text processing and system interaction. Characterized by higher-layer abstractions and flexibility. Examples include sh, perl, Python, and Ruby.

Other Important Language Features

The sources also mention various other important language features, including:

- Regular expression handling

- Objects (and inheritance)
- Closures/code blocks
- Immutability
- Tail recursion
- Pattern matching (and unification)
- Abstract types
- Garbage collection
- Declarations (explicit and implicit)
- Type systems (static, polymorphic, dynamic, type safety)

Language Implementation

There are two main approaches to implementing a programming language:

- **Compilation:** The source program is translated into another language, often directly executable machine code.
- **Interpretation:** The interpreter executes each instruction in the source program step-by-step without creating a separate executable.

Compilers offer efficiency, while interpreters are helpful for debugging and have a faster start time.

Attributes of a Good Language

A good language is typically characterized by:

- Low cost of use (including runtime, translation, creation, and maintenance).
- Portability (the ability to run on different systems).
- A supportive programming environment (libraries, documentation, community, IDEs, etc.).
- Clarity, simplicity, and unity (a consistent framework for expressing algorithms).
- Orthogonality (meaningful feature combinations and independent operation).

- Naturalness for the application (program structure reflects the problem's logic).
- Support for abstraction (hiding unnecessary details).
- Security and safety (making it hard to write unsafe code).
- Ease of program verification (ensuring correctness).

Summary