Language Design & Implementation – 6CC509

Lecture 1: Introduction

Ioannis Tsioulis

Introduction to the module

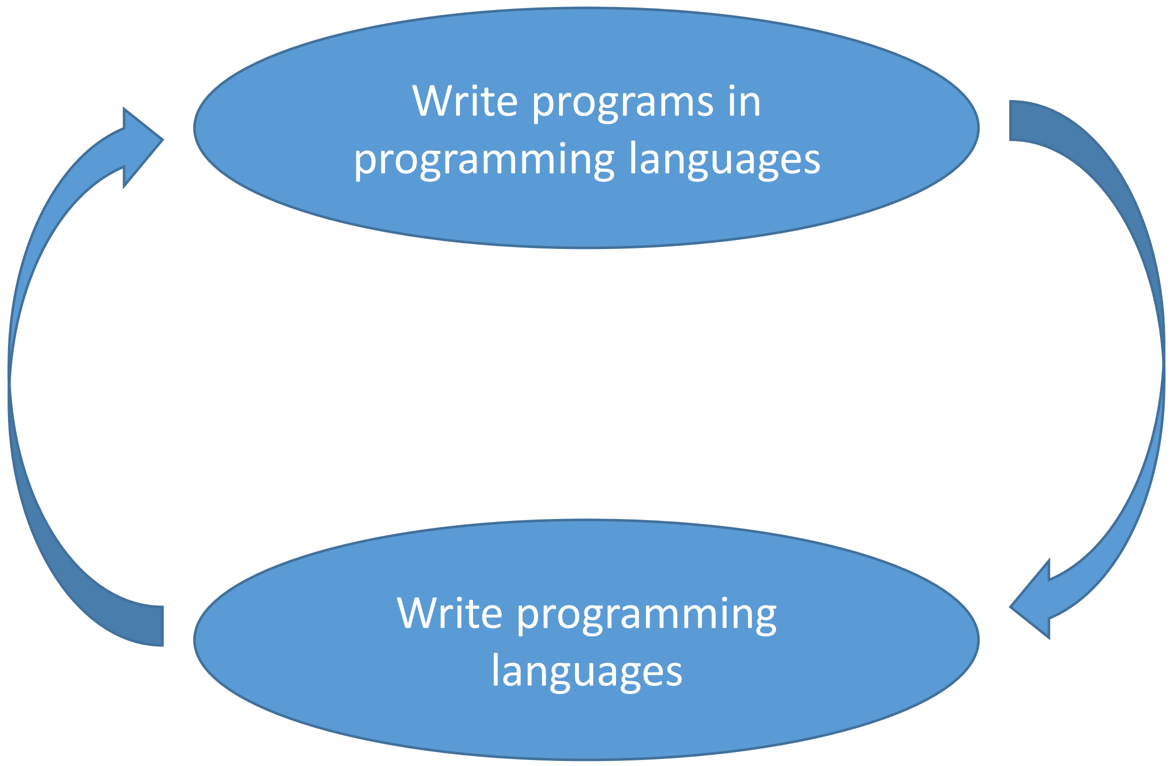
What is This Module About?

- Learning Computer Science is about…
  - Understanding computational “machinery”
    - Computational math
    - Algorithms
    - Data structures
    - Graphics
    - Networks & distributed systems
    - Databases
    - Operating systems
  - Taking control of computational machinery
    - All of the above, and…
      - Programming
      - Software engineering

This gives you knowledge of the machine, which gives you power over the machine.

Why LDI?

The first power you were given…


Write programs in programming languages
Write programming languages

The final power you need…

…to complete the circle.

What is This Module About?

"Computer languages are a cornerstone of practical and theoretical applications of computer science. Therefore, this module provides an in-depth and pragmatically-focused exploration of the theory, concepts, paradigms, current research, and practical implementation issues involved in designing and implementing mark-up languages, special-purpose languages, and programming languages."

Module Learning Outcomes

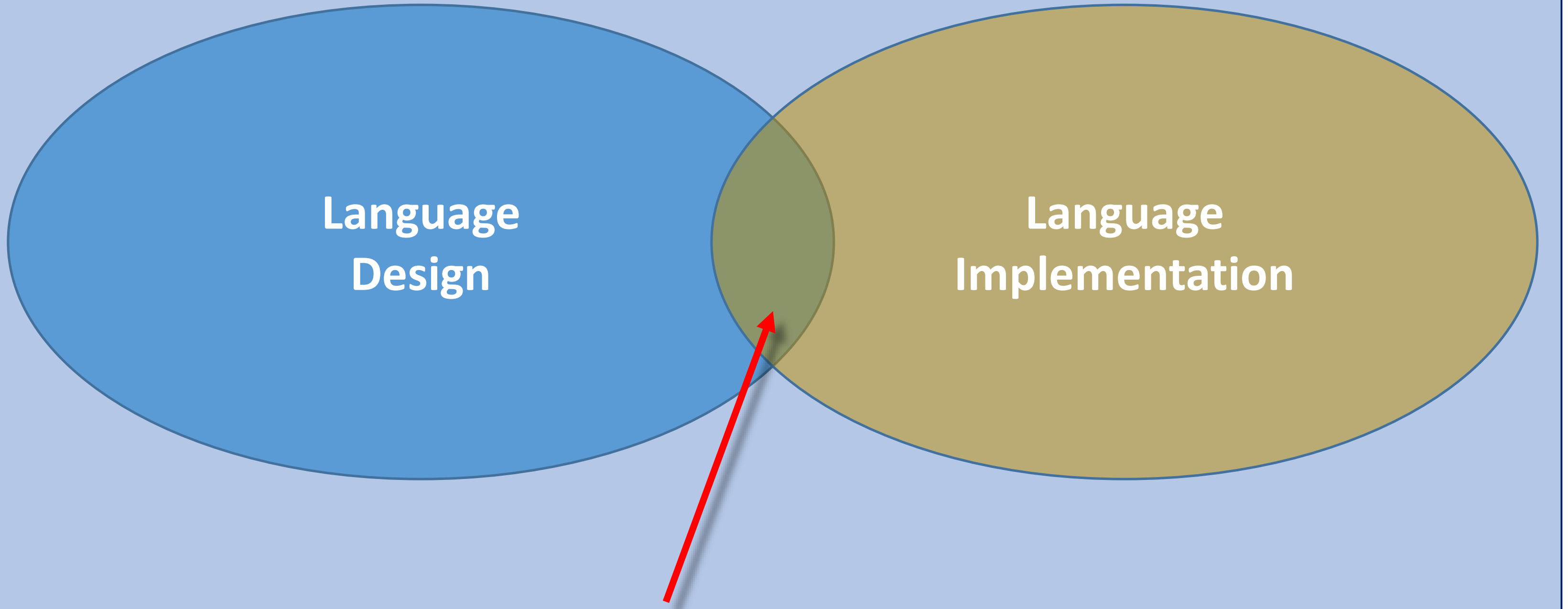
On successful completion of the module, students will be able to:

1. Demonstrate a critical awareness of the theory and practical issues involved in implementing language parsers, compilers and interpreters.

2. Design a computer language and be able to implement significant portions of an interpreter and/or compiler for it.

What’s this module *really* about?

- Learn to design and implement computer languages.
  - Markup languages
  - DSLs
  - Programming languages
  - Query languages
  - Etc.
- Two Modules in One



This intersection is infinitely large, but infinitesimally small compared to the sizes of the intersecting sets.

How?

- Largely <u>informal</u>, practical, and <u>not</u> theoretical.
  - Hopefully fun & useful.
  - Enough introduction to set you on the right path to explore further.
- Structured debates, to explore language design.
- Conventional lectures, to introduce language implementation.
- Learn-by-doing + learn-by-independent-study.

Topics

Dynamic vs Static Typing

The proponents of dynamic typing argue that it is more flexible and abstract than static typing. The proponents of static typing argue that dynamic typing misses errors that you can catch with static typing, and the more errors you can catch at compile-time, the fewer users will have to endure at run-time. You can only choose one. Which one?

Take out the Garbage Yourself? Or get Someone Else to Do It?

Automated garbage collection – algorithmically freeing memory that is no longer in use – has been a feature of languages like FORTRAN and LISP since their inception, and is still found in modern general-purpose programming languages like C# and Java. Languages like C and C++ don't do this – they put the responsibility for memory management squarely in the hands of the developer. Proponents of C and C++ argue that this is the way it should be done because automated garbage collection is slow and inefficient. Proponents of automated garbage collection argue that manual memory management is a source of memory leaks and unnecessary development effort. Do you agree?

Object Oriented Programming vs Functional Programming

Thirty years ago, Object Oriented programming was primarily a research topic, mainly employed tentatively on experimental projects. The professional developer community largely regarded it with fear, uncertainty, and doubt. Now, of course, it is mainstream. However, some pundits argue that Functional Programming – which is a completely different paradigm from procedural and object-oriented programming (both of which belong to the imperative paradigm) – is in the same position Object Oriented programming was three decades ago, and is now poised to replace it. Do you agree?

You Decide!

Think about topics – related to language design – that you'd like to debate and we'll agree on a topic.

Topic Areas

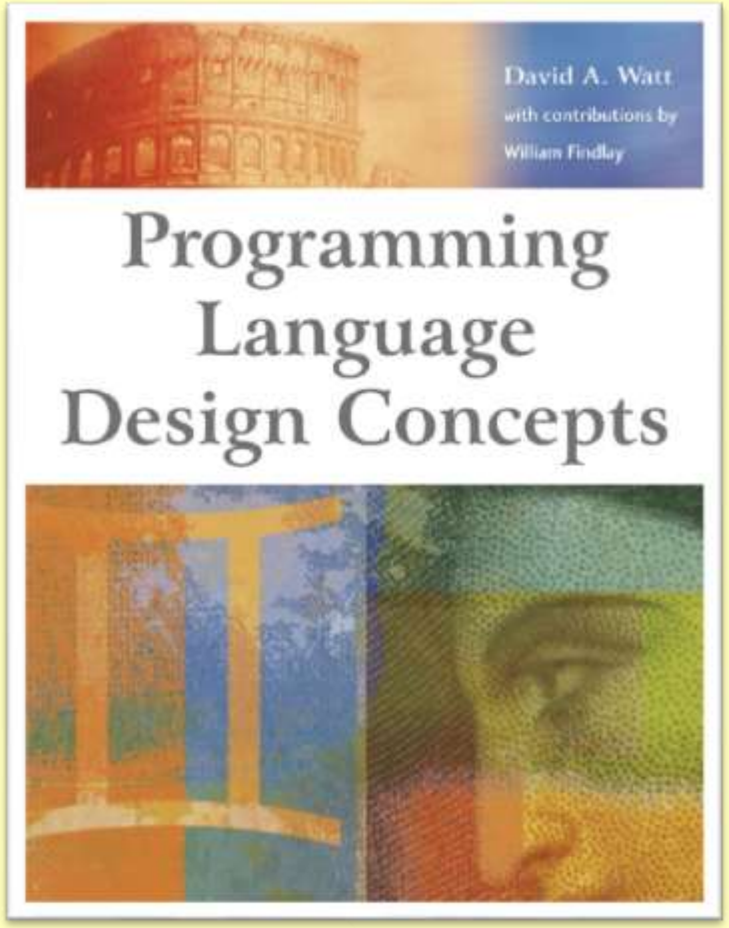
- Models of Computation
  - Imperative
  - Functional
  - Logical
- Language paradigms
  - Procedural
  - Functional
  - Logical
  - Declarative
  - Object-oriented
- **Language design**
  - **Syntax**
  - **Semantics**
  - **Type systems**

Introduced via conventional lecture; Explored via structured debate & Independent Study.

- **Language implementation**
  - **Parsing**
  - **Grammars**
  - **Optimisation**
  - **Compiler construction**
  - **Cross-compilation**
  - **Virtual and real machines**

Introduced via conventional lectures; Explored via Independent Study and tutorial exercises.

Course Textbook …

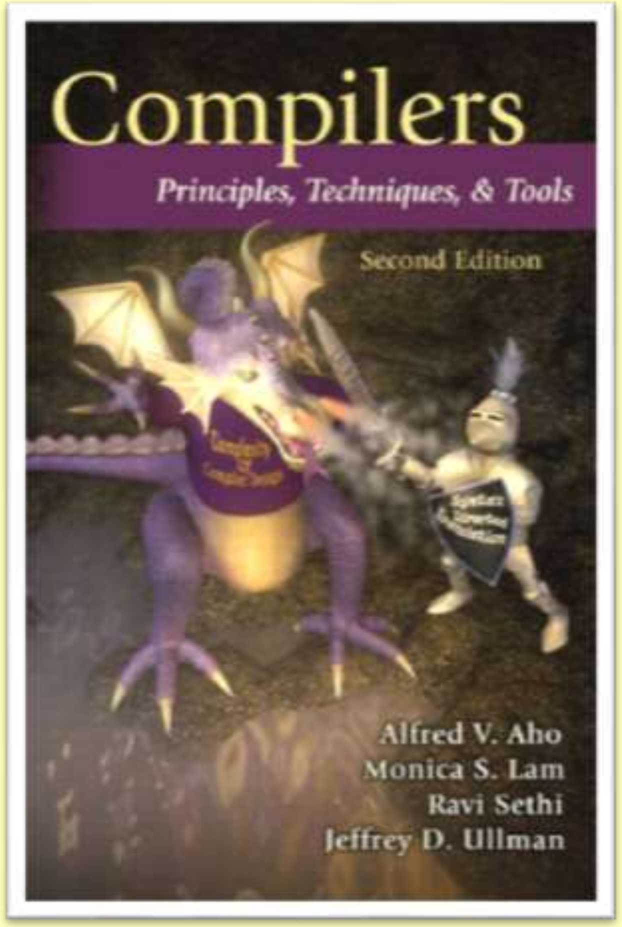


D. Watt (2004), *Programming Language Design Concepts*, Wiley.

This book explains the concepts underlying programming languages and demonstrates how these concepts are synthesized in the major paradigms: imperative, OO, concurrent, functional, logic and scripting.

Copies available

Highly Recommended Reading …

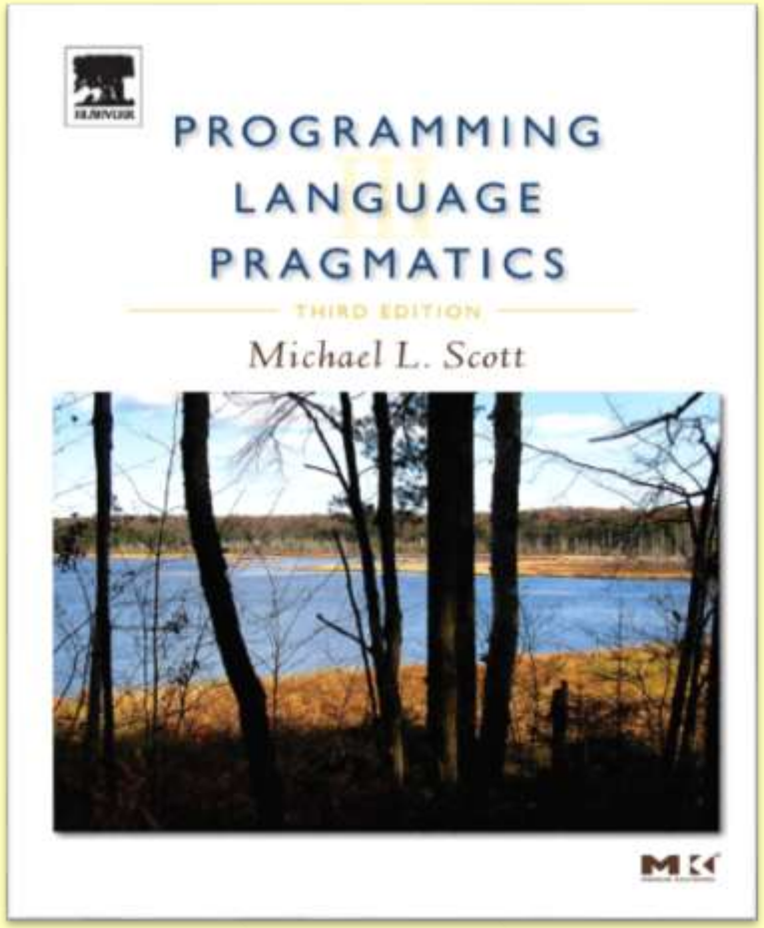


A. Aho, M. Lam, R. Sethi and J. Ullman (2006), *Compilers: Principles, Techniques, & Tools*, 2nd Ed., Addison-Wesley.

This book is known to professors, students, and developers worldwide as the "Dragon Book". Every chapter has been revised to reflect developments in software engineering, programming languages, and computer architecture that have occurred since 1986, when the last edition published.

Copies available

Other Recommended Reading …

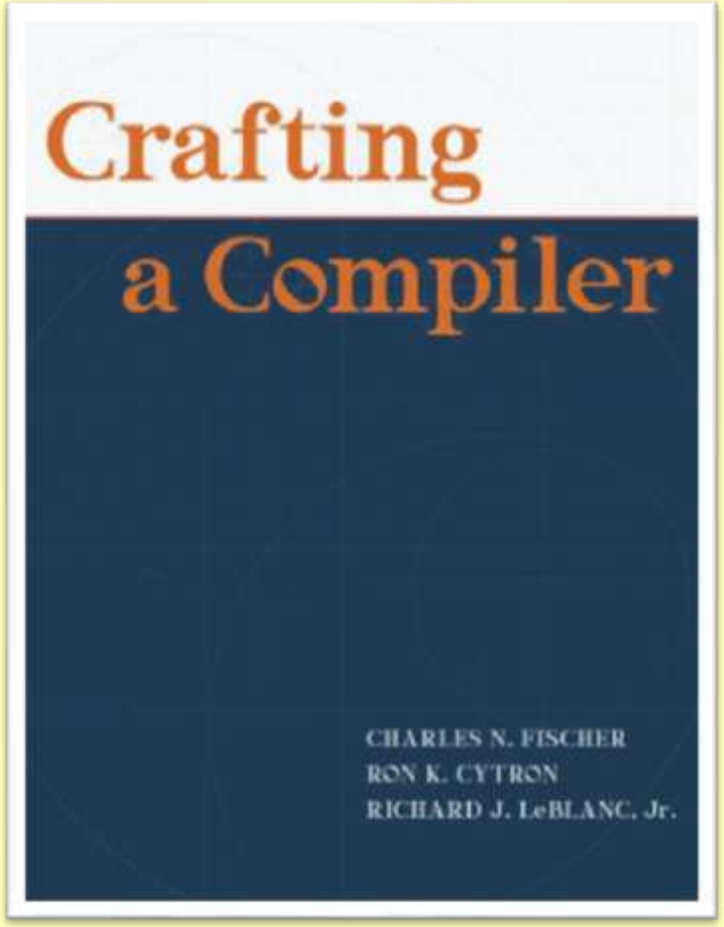


M. Scott (2009), *Programming Language Pragmatics*, 3d Ed., Morgan Kaufmann

Taking the perspective that language design and implementation are tightly interconnected and that neither can be fully understood in isolation, this book covers the most recent developments in programming language design and implementation.

Copies available

Other Recommended Reading …

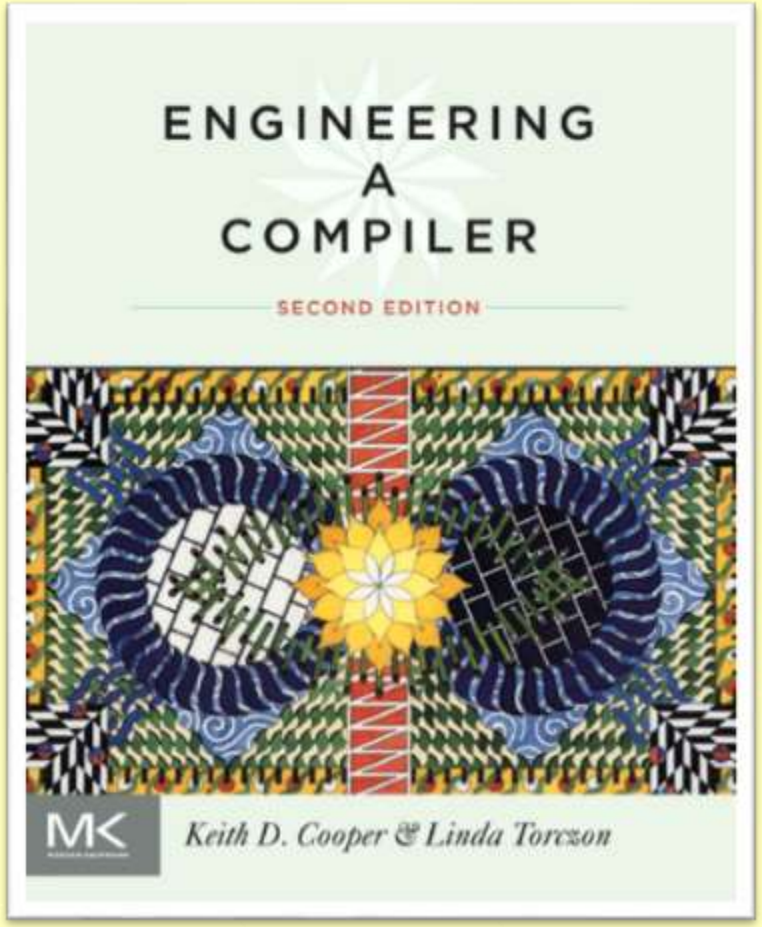


C. Fischer, R. Cytron and R. LeBlanc (2010), *Crafting a Compiler*, Addison-Wesley.

Crafting a Compiler is an undergraduate-level text that presents a practical approach to compiler construction with thorough coverage of the material and examples that clearly illustrate the concepts in the book.

Copies available

Other Recommended Reading …

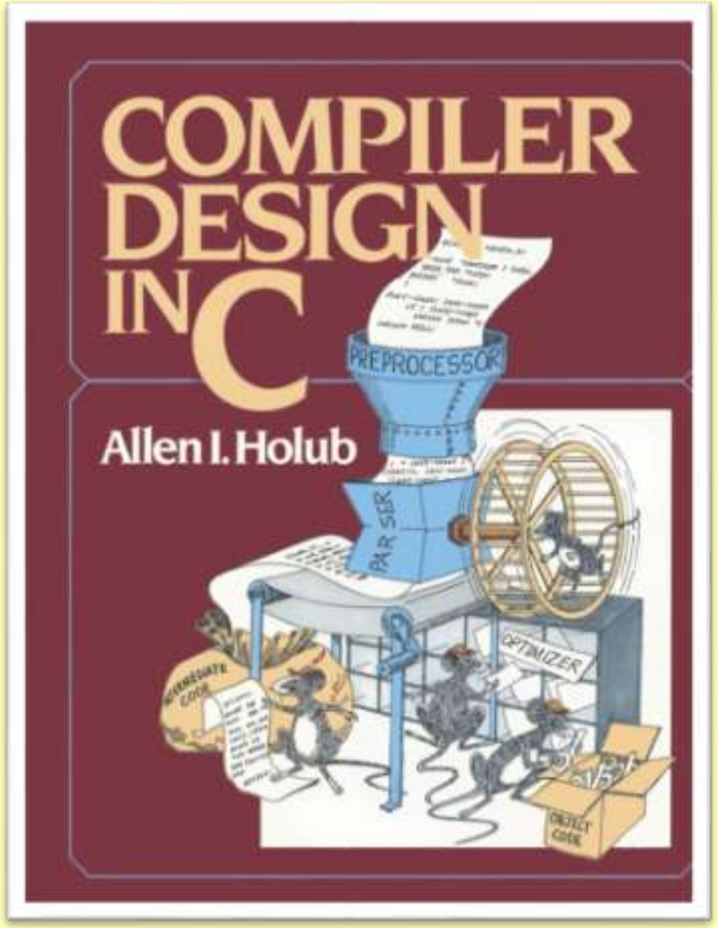


K. Cooper and L. Torczon (2012), *Engineering a Compiler*, 2nd Ed., Elsevier.

In this comprehensive text you will learn important techniques for constructing a modern compiler. It combines basic principles with pragmatic insights for building state-of-the-art compilers.

Copies available

Implementation Textbook …

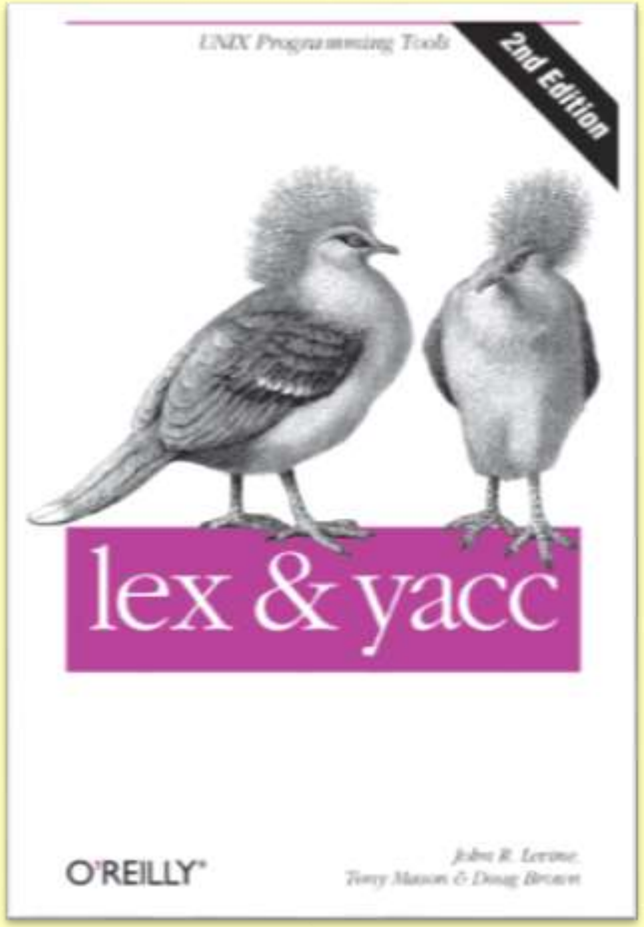


A. Holub (1990), *Compiler Design in C*, Prentice Hall.

This book presents the subject of Compiler Design in a way that's understandable to a programmer, rather than a mathematician. The best way to learn how to write a compiler is to look at one in depth; the best way to understand the theory is to build tools that use that theory for practical ends.

Copies available

Other Implementation Textbook …



J. Levine, T. Mason and D. Brown (1992), *lex & yacc*, 2nd Ed., O' Reilly.

This book shows you how to use two utilities, lex and yacc, in program development. These tools help programmers build compilers and interpreters, but they also present a wider range of applications.

Copies available

Introduction to Language Design

What are some of the design decisions?

- Domain-specific or general-purpose?
- Functional or imperative/procedural or logic-deductive or mixed-paradigm?
- Turing-complete or not?
- Declarative or imperative?
- Dataflow / stream processing?
- C-derived syntax, or something else?
  - Infix, prefix, postfix?
  - Text or graphical?
  - Simple grammar vs complex grammar.
- Type system model?
  - Dynamically-typed or statically-typed?
  - Manifest typing or type inference?
  - Weakly-typed or strongly-typed?
  - Algebraic types? E.g., sum of products?
- Modularization? Namespaces?
- Functions/procedures/operators/predicates?
- Some sort of object-orientation? Or not?
  - Inheritance? Encapsulation? Polymorphism? Templates? Interfaces? Substitutability?

- Constraints?
- Lambdas, closures, continuations?
- First-class… Functions, threads, control structures, variables, whatever?
- Variables: Mutable or immutable?
- Built-in arrays vs built-in containers vs library containers?
- Built-in I/O vs I/O via library operators
- Operator overloading? Operator overriding?
- Automatic garbage collection or explicit memory management?
- Lazy evaluation (and memorization?) vs eager evaluation?
- Pointers vs References vs Value Semantics
  - Function pointers?
- Pass by reference/value/name?
- Comments that do special things?
  - See literate Haskell.
  - Annotations? See Java.
- Esoteric or conventional?
- Interpreter or compiler or transpiler?

Understand the Distinction: Computational vs Language Paradigms

- Models of Computation (aka Computational Paradigms)
  - Turing Machines
  - Lambda Calculus
  - Cellular Automata
  - Combinatory logic
  - Etc.
- Language Paradigms
  - Some say there are three known, and one yet to be discovered (or at least implemented):
    - http://wiki.c2.com/?ThereAreExactlyThreeParadigms
  - Some say there are many:
    - https://en.wikipedia.org/wiki/Programming_paradigm
  - Some say there are none:
    - http://wiki.c2.com/?ThereAreNoParadigms

Understand the Distinction: Procedural vs Functional

- A language isn't "functional" just because it has a construct called 'function'.
- C is not a functional programming language, though it might be used to build functional programming languages.
- JavaScript is closer to being a functional programming language than C.
- It's helpful to understand why. (Homework!)

Understand the Distinction: Syntax vs Semantics

- Language = syntax + semantics

What you can say.

a = 3;

What you say *does*. (Literally, what it *means.)*

"The value denoted by evaluating the literal expression '3' is assigned to the variable identified by the name 'a'."

Understand the Distinction: Imperative vs Declarative

- Imperative
  1. Do this first.
  2. Then do that.
  3. Do the other thing last.

- Declarative
  - There are three things.

Understand the Distinction: Languages vs IDEs or Editors

- Visual Studio
- Emacs
- Eclipse
- Netbeans
- IntelliJ
- Notepad++

Editors/IDEs (generally) aren't languages.

They're usually not interesting in language terms, but interesting Editor/IDE features might be language dependent.

Test: Can the language exist and be used without these?

There are visual programming languages *entirely* dependent on a graphical environment.

Despite the name, Visual Studio isn't one of them.

Understand the Distinction: Languages vs Libraries

Is a **language standard library**…

- E.g. the C standard library (libc), C++ STL, C++ Standard Library, .NET Framework, Java Platform, etc.

…part of the language?

Understand the Distinction: Dynamically vs Statically typed

Handy rule of thumb:

- **Statically-typed languages** tend to associate types with variables and values (expressions). A value (expression) can only be assigned to a variable if their types are type compatible, and this can often be determined (at least to some degree) at compile-time.

- **Dynamically-typed languages** tend to associated types only with values (expressions). A value (expression) can be assigned to a variable unconditionally.

Understand the Distinction: Single vs Multiple-dispatch

- ***Single dispatch*** = conventional OO
  - Dynamically dispatch on the run-time type (as opposed to the declared or static type) of the (often implicit or invisible) first argument.
- ***Multiple dispatch*** = more powerful than conventional OO
  - Dynamically dispatch on the run-time type (as opposed to the declared or static type) of all arguments.
- Other forms of dispatch are possible…

Understand the Distinction: First vs Second-class

- Usually ...
  - ***First-class*** constructs can be defined by expressions and assigned to (or bound to) variables or parameters at run-time and returned by operators/functions.
  - ***Second-class*** constructs can’t.

Understand the Distinction: Mutable vs Immutable

- ***Mutable*** things can change. ***Immutable*** things can't.
  - Functional programming derives considerable benefit from avoiding mutability.
  - For example, if *variables can be assigned once and only once*, programs can become much easier to reason about.

Study These Languages

- Lisp
- Fortran
- COBOL
- ALGOL
- BASIC
- Pascal
- Logo
- Ada
- APL
- Forth
- Prolog
- Erlang
- Ruby
- Haskell
- Squeak (Smalltalk)
- Scratch
- Toontalk
- Go
- Rust
- Swift
- D

You should already be familiar with these:

- C
- C++
- C#
- PHP
- Java
- JavaScript
- SQL
- Python

Suggested Reading

- Why does the world need more programming languages?
  - https://www.fastcompany.com/3031443/why-does-the-world-need-more-programming-languages
- Interactive language influence network:
  - https://exploringdata.github.io/vis/programming-languages-influence-network/
- The programming languages weblog:
  - http://lambda-the-ultimate.org/
- Interview with Alan Kay:
  - http://www.drdobbs.com/architecture-and-design/interview-with-alan-kay/240003442?pgno=1
- “The Dragon Book”
  - https://suif.stanford.edu/dragonbook/

MEDITERRANEAN COLLEGE

Resources

Bibliography

Sources for the preparation of the lectures' slides:

- D. Watt (2004), *Programming Language Design Concepts*, Wiley.
- M. Scott (2009), *Programming Language Pragmatics*, 3d Ed., Morgan Kaufmann
- A. Aho, M. Lam, R. Sethi and J. Ullman (2006), *Compilers: Principles, Techniques, & Tools*, 2nd Ed., Addison-Wesley.
- C. Fischer, R. Cytron and R. LeBlanc (2010), *Crafting a Compiler*, Addison-Wesley.
- K. Cooper and L. Torczon (2012), *Engineering a Compiler*, 2nd Ed., Elsevier.
- A. Holub (1990), *Compiler Design in C*, Prentice Hall.
- J. Levine, T. Mason and D. Brown (1992), *lex & yacc*, 2nd Ed., O' Reilly.
- Voorhis, D. (2019), *Language Design and Implementation*, Slides Presentation, University of Derby
- CSD Univ. of Crete (2012), *Programming Languages & Paradigms*, Slides Presentation, CSD Univ. of Crete

MEDITERRANEAN COLLEGE

Thank You!