CSci 658-01: Software Language Engineering
Spring 2018 Lecture Notes

H. Conrad Cunningham

4 May 2018

Contents

Professor of Computer and Information Science
University of Mississippi
211 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-5358

I maintain these notes as text in the Pandoc's dialect of Markdown using

embedded LaTeX markup for the mathematical formulas and then translate the notes to HTML, PDF, and other formats as needed.

Advisory: The HTML version of this document requires use of a browser that supports the display of MathML. A good choice as of May 2018 is a recent version of Firefox from Mozilla.

Lecture Notes

Go To Current Lecture

Lecture Notes, Examples, and Schedule

Language-Oriented Programming

1. **(22 Jan) Review syllabus.** Distribute papers to be discussed over next two weeks.
2. **(24-26 Jan) Discuss "Little Languages" paper**
   a. Paper: Jon Bentley. Programming Pearls: Little Languages, *Communications of the ACM*, Vol. 29, No. 8, pp. 711-721, August 1986.
   b. Software: The `groff` package includes the GNU implementations of the `troff` set of "Unix" tools such as `pic` and `chem`. (I installed this on my iMac using Homebrew. I also installed the `plotutils` package to help with display in various formats.)
   c. Additional references for little language `pic` (not discussed)
      - Brian W. Kernighan. PIC -- A Graphics Language for Typesetting, Revised User Manual, Computing Science Technical Report No. 116, Bell Laboratories, December 1984. [local]
      - Philipp K. Janert. In Praise of Pic, *ONLamp.com*, O'Reilly Media, June 2007, Retrieved 24 January, 2018.
      - Eric S. Raymond. Making Pictures with GNU PIC, August 1995.
      - W. Richard Stevens. Examples of `pic` Macros, Retrieved from http://www.kohala.com/start/ (troff resources, pic), 24 January 2018. [`pic` source]
   d. Additional references for the little languages `lex`, `yacc`, `make`, and `chem` (not discussed)
      - Jon L. Bentley, Lynn W. Jelinski, and Brian W. Kernighan. Chem – A Program for Phototypesetting Chemical Structure Diagrams, AT&T Bell Laboratories, Murray Hill, NJ 07974, U.S.A.

- Stuart I. Feldman. Make – A Program for Maintaining Computer Program, *Software: Practice and Experience*, Vol. 9, no. 4 pp. 255-265, 1979.
- Stephen C. Johnson. Yacc: Yet Another Compiler-Compiler, Vol. 32. Murray Hill, NJ: Bell Laboratories, 1975.
- Michael E. Lesk and Eric Schmidt. Lex: A Lexical Analyzer Generator, Bell Laboratories Murray Hill, NJ, 1975.
- Learn X in Y Minutes, Where X = make (GNU)
- Learn X in Y Minutes, Where X = bash (GNU)

3. **(26-29 Jan) Discuss "No Silver Bullet" paper** and related issues

   a. Paper: Frederick P. Brooks. No Silver Bullet: Essence and Accident in Software Engineering, *IEEE Computer*, Vol. 20, No. 4, 10-19, 1987.

   b. Retrospective paper after 10 years (not discussed): Frederick P. Brooks. 'No Silver Bullet' Refired, Chapter 17 in *The Mythical Man Month: Essays on Software Engineering, Anniversary Edition*, Addison Wesley, 1995.

4. **(29-31 Jan) Discuss "Language-Oriented Programming" paper**

   a. Paper: M. P. Ward. Language-Oriented Programming, *Software–Concepts and Tools,* Vol. 15, No. 4, pp. 147-161, 1994. [local]

   b. Additional references from paper (not discussed)

      - Charles Antony Richard Hoare. The Emperor's Old Clothes (1980 Turing Award Address), *Communications of the ACM*, Vol. 24, No. 2, 75-83, 1981.

5. **(31 Jan; intermittently until Spring Break, 19-28 Mar; 2-6 Apr) Discuss overview of Domain Specific Languages**

   a. Instructor's notes on Domain Specific Languages

   b. Mixed-in introductory discussion of Fowler's DSL book introductory chapters, State Machine DSL, and Computer Configuration DSL below.

   c. References used in notes (most not discussed directly)

      - Martin Fowler. Using Domain-Specific Languages, Chapter 2, *Domain-Specific Languages*, Addison Wesley, 2011.
      - Steve Freeman and Nat Pryce. Evolving an Embedded Domain-Specific Language in Java, In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 12 pages, 2006. [local]
      - Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler and Steven Voelkel. Design Guidelines for

Domain Specific Languages, In *Proceedings of OOPSLA Workshop on Domain-Specific Modeling*, 2009. Also arXiv preprint arXiv:1409.2378, 2014. [local]

- Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and How to Develop Domain Specific Languages, *ACM Computing Surveys*, 37(4):316-344, December 2005. [local]
- Scott A. Thibault, Renaud Marlet, and Charles Consel. Domain-Specific Languages: From Design to Implementation–Application to Video Device Drivers Generation. *IEEE Transactions on Software Engineering*, 25(3):363–377, May/June 1999.

6. **(dates on items) Survey Design Patterns**

   a. **(5-7 Feb)** Instructor's notes on Introduction to Patterns [HTML slides]

   b. **(7 Feb)** Instructor's notes on Pipes and Filters Architectural Pattern (Not fully covered, used mainly to illustrate approach to pattern definition) [Powerpoint]

   c. **(9-12 Feb)** John Vlissides. Designing with Patterns, In *Pattern Hatching: Design Patterns Applied*, Addison-Wesley, 1998. [Powerpoint]

   d. Additional references (not directly discussed in class)

      - Source Making Design Patterns catalog
      - Instructor's slides on Factory Method Design Pattern (Powerpoint)
      - Instructor's slides on Strategy Design Pattern (Powerpoint)
      - Instructor's slides on Template Method Design Pattern (Powerpoint)
      - [Siemens book] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*, Wiley, 1996.
      - ["Gang of Four" (GoF) book] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995. Patterns*, Wiley, 1996.
      - Mary Shaw. Some Patterns for Software Architecture, In John M. Vlissides, James O. Coplien, and Norman L. Kerth, editors, *Pattern Languages of Program Design 2*, Addison Wesley, 1996, pages 255-270. [local]

7. **(5, 14-21 Feb) Discuss Fowler's introductory DSL example**

   a. Martin Fowler. An Introductory Example, Chapter 1, *Domain-Specific Languages*, Addison Wesley, 2011.

b. Martin Fowler. Using Domain-Specific Languages, Chapter 2, *Domain-Specific Languages*, Addison Wesley, 2011.

External DSLs

8. **(dates on items) Discuss State Machine External DSLs** based on Martin Fowler's Secret Panel Controller (State Machine) DSLs

   a. **(2-5, 14-26 Feb)** Martin Fowler. *Domain Specific Languages*, Addison Wesley, 2011.

      - Fowler introduces the running example, the Secret Panel Controller (State Machine) case study, in DSL Chapter 1)
      - Fowler DSL Chapter 1 "An Introductory Example"
      - Fowler DSL Chapter 2 "Using Domain-Specific Languages"
      - Fowler DSL Chapter 3 "Implementing DSLs"
      - Fowler DSL Chapter 5 "Implementing an External DSL"
      - Fowler DSL Chapter 8 "Code Generation"
      - Fowler DSL Chapter 11 "Semantic Model"
      - Fowler DSL Chapter 17 "Delimiter-Directed Translation"
      - Fowler DSL Chapter 18 "Syntax-Directed Translation"
      - Fowler DSL Chapter 21 "Recursive Descent Parser"
      - Fowler DSL Chapter 21 "Parser Generator"
      - Fowler DSL Chapter 22 "Parser Combinator"
      - Fowler DSL Chapter 51 "State Machine"
      - Fowler DSL Chapter 52 "Transformer Generation"
      - Fowler DSL Chapter 55 "Model Aware Generation"
      - Fowler DSL Chapter 56 "Model Ignorant Generation"
      - Fowler's List of DSL Patterns from DSL book.

   b. Other items from the instructor's notes

      - Mealy Machine Description and Exercise gives a formal description of this kind of state machine
      - Recursive Descent Parsing gives a set of (Haskell) program templates for recursive descent parsers
      - Parsing Combinators describes a preliminary set of (Haskell) parsing combinators for recursive descent parsers
      - Wikipedia entry on Parsing Combinator

      Scala versions (2009)

   c. Scala notes

      - I developed the Scala versions in 2009 by referring to an in-work version of Fowler's book on his website, so they may not follow his final book precisely. Fowler's code is in Java, Ruby, C#, etc.

I updated the Scala programs in 2018 to make sure they would compile and execute under the current Scala release.

- Scala references below

d. **(21 Feb)** Graphviz and `dot` language reference and examples (looked at examples in class)

- Note: On my iMacs, I used Homebrew to install the package `graphviz`.
- Graphviz: Graph Visualization Software and the little language/tool `dot`
- `ExprLangModDep.gv` – `script` – `png` output
- `DFA01.gv` – script – `pdf` output
- `NFA01.gv` – script – `svg` output
- `NFA01a.gv` – script – `svg` output
- `Infix1plus1b.gv` – script – `png` output

e. **(14 Feb)** Scala State Machine Semantic Model (Ch. 1, 3, 11)

- StateMachine.scala
- StateMachineTest.scala
- CommandChannel.scala (mock)
- StateMachineDirect.sh test script

f. **(21 Feb)** Scala XML-based External DSL (Ch. 1, 3, 5)

- StateMachineXMLTest .scala
- IncrementalStateMachineBuilder.scala
- input file SecretPanel.xml
- StateMachineXML.sh test script

g. **(16 Feb)** Scala Custom External DSL with a Delimiter-Directed parser (Ch. 1, 3, 5, 17)

- DelimiterDSLTest.scala
- IncrementalStateMachineBuilder.scala
- input file CustomExternalStateMachineDSL.dsl
- DelimiterDSL.sh test script

h. **(19 Feb)** Scala Custom External DSL with hand-coded Ad Hoc Recursive-Descent parser (Ch. 1, 3, 5, 21)

- RecursiveDescentTest.scala
- IncrementalStateMachineBuilder.scala
- input file CustomExternalStateMachineDSL2.dsl
- RecursiveDescent.sh test script

i. **(21 Feb briefly)** Scala Custom External DSL (using Scala parser combinator library) with embedded state machine builder (Ch. 1, 3, 5, 22)

- CombinatorParserBuilderTest.scala
- IncrementalStateMachineBuilder.scala
- input file CustomExternalStateMachineDSL2.dsl
- CombinatorParserBuilder.sh test script

j. **(21 Feb briefly)** Scala Custom External DSL (using Scala parser combinator library) with full AST construction (Ch. 1, 3, 5, 22)

- CombinatorParserASTTest.scala
- input file CustomExternalStateMachineDSL2.dsl
- CombinatorParserAST.sh test script

k. **(21 Feb briefly)** Scala Static C Code Generator with Model-Aware target platform library (Ch. 1, 3, 5, 8, 52, 55)

- StaticC_GeneratorTest.scala
- StaticC_Generator.sh test script
- generated C output file output.c
- Note: The needed framework code needed to run the generated C program is not yet available in this form. It needs to be reconstructed from Fowler's book.

l. **(21 Feb)** Scala Graphviz Dot Language Code Generator (added 2018) (Ch. 1, 3, 5, 8, 52)

- GraphVizCodeGen.scala
- GraphVizCodeGenTest.scala
- GraphVizCodeGen.sh test script
- generated Graphviz Dot output file graph.gv (see test script)
- Scalable Vector Graphics output from dot graph.svg (see test script)
- PDF output from dot graph.pdf (see test script)

9. **(16 Feb) Announce homework Assignment #1 and Assignment #2**

10. **(23 Feb) Discuss Metaprogramming**

    Note: We return to this topic near the end of the semester with an emphasis on Python 3 metaprogramming features.

Internal DSLs

11. **(dates on items) Discuss Fowler's Computer Configuration Internal DSLs** (Ch. 4, 35, 36, 38)

    a. Background: Martin Fowler. *Domain Specific Languages*, (Addison Wesley, 2011).

- Martin Fowler's List of DSL Patterns from *Domain Specific Languages*, Addison Wesley, 2011
- Fowler DSL Chapter 1 "An Introductory Example"
- Fowler DSL Chapter 2 "Using Domain-Specific Languages"
- Fowler DSL Chapter 3 "Implementing DSLs"
- Fowler DSL Chapter 4 "Implementing an Internal DSL"
- Fowler DSL Chapter 11 "Semantic Model"
- Fowler DSL Chapter 13 "Context Variable"
- Fowler DSL Chapter 32 "Expression Builder"
- Fowler DSL Chapter 33 "Function Sequence"
- Fowler DSL Chapter 34 "Nested Functions"
- Fowler DSL Chapter 36 "Object Scoping"
- Fowler DSL Chapter 37 "Closure"
- Fowler DSL Chapter 38 "Nested Closure"
- Fowler DSL Chapter 39 "Literal List"
- Fowler DSL Chapter 40 "Literal Map"

Scala versions (2009)

b. **(23 Feb)** Scala semantic model (shared)

c. **(23, 26 Feb)** Scala internal DSL using Method Chaining

d. **(26 Feb)** Scala internal DSL using Nested Closures and Object Scoping

e. CompConfig.sh test script

12. **(26 Feb) Discuss Fowler's Email Message Building Internal DSL** (Ch. 35)

    a. Background: Martin Fowler. *Domain Specific Languages*, (Addison Wesley, 2011). See listing of chapters under Computer Configuration internal DSL above.

    Scala versions (2009)

    b. Scala internal DSL using Method Chaining and Progressive Interfaces

    c. EmailProgressive.sh test script

13. **(see dates on items) Explore Fowler's Lair Configuration DSLs**

    a. Background on Lair DSL

    - **(28 Feb; 2-7 Mar)** Case study: Martin Fowler's One Lair and Twenty Ruby DSLs, Chapter 3, *The ThoughtWorks Anthology: Essays on Software Technology and Innovation*, The Pragmatic Bookshelf, 2008 [local copy]
    - Source code: [entire Thoughtworks book] [Fowler's Ruby DSLs]
    - Patterns (repeated from above): Martin Fowler's List of DSL Patterns from *Domain Specific Languages*, Addison Wesley, 2011

b. Background on internal DSLs: Martin Fowler. *Domain Specific Languages*, (Addison Wesley, 2011). See chapters under Computer Configuration internal DSL above.

c. Background on Lua

- See Lua references below
- Several of the Lair DSL programs must execute with Lua 5.1 because they use a few Lua platform features changed in Lua 5.2 or 5.3. I installed the lua@5.1 package on my iMacs using the Homebrew package manager.
- At the terminal command line, `lua5.1 testNN.lua` compiles and executes the test driver program for the DSL with files builder program `builderNN.lua` and DSL script `rulesNN.lua`.

d. **(28 Feb Python)** Shared modules

Fowler's Ruby source

- Semantic model (model.rb)
- Test Driver for semantic model (rules0.rb)

Lua (2013)

- Class support module (for implementing classes in Lua)
- Semantic model
- Test driver for semantic model (rules00.lua)

Python 3 (2018)

- Semantic model
- Test driver for semantic model (rules00.py)

e. **(2 Mar Python)** Internal DSL using Global Function Sequence pattern

Fowler's Ruby

- builder module (builder8.rb)
- dsl script (rules8.rb)

Lua (2013)

- builder module (builder08.lua)
- dsl script (rules08.lua)
- test driver (test08.lua)

Python 3 (2018)

- builder module (builder08.py)
- direct execution test of DSL script (rules08x.py)
- dynamically loaded dsl script (rules08.py)
- test driver for dynamically loaded dsl script (test08.py)

f. **(7 Mar Python)** Internal DSL using Class Method Function Sequence and Method Chaining patterns

   Fowler's Ruby

   - builder module (builder11.rb)
   - dsl script (rules11.rb)

   Lua (2013)

   - builder module (builder11.lua)
   - dsl script (rules11.lua)
   - test driver (test11.lua)

   Python 3 (2018)

   - builder module (builder11.py)
   - dynamically loaded dsl script (rules11.py)
   - test driver for dynamically loaded dsl script (test11.py)

g. **(7 Mar Python)** Internal DSL using Expression Builder and Method Chaining patterns

   Fowler's Ruby

   - builder module (builder14.rb)
   - dsl script (rules14.rb)

   Lua (2013)

   - builder module (builder14.lua)
   - dsl script (rules14.lua)
   - test driver (test14.lua)

   Python 3 (2018)

   - builder module (builder14.py)
   - dynamically loaded dsl script (rules14.py)
   - test driver for dynamically loaded dsl script (test14.py)

h. Internal DSL using Nested Closures pattern

   Fowler's Ruby

   - builder module (builder3.rb)
   - dsl script (rules3.rb)

   Lua (2013)

   - builder module (builder03.lua)
   - dsl script (rules03.lua)
   - test driver (test03.lua)

Python 3 – none yet. (Python's weak syntactic support for lambdas does not allow the relatively direct approach usable in Ruby, Scala, and Lua.)

i. Internal DSL using Expression Builder, Object Scoping, and Method Chaining patterns

Fowler's Ruby

- builder module (builder17.4b)
- dsl script (rules17.rb)

Lua (2013)

- builder module (builder17.lua)
- dsl script (rules17.lua)
- test driver (test17.lua)

Python 3 – none yet

j. Internal DSL using Literal Collection pattern

Fowler's Ruby

- builder module (builder22.rb)
- dsl script (rules22.rb)

Lua (2013)

- builder module (builder22.lua)
- dsl script (rules22.lua)
- test driver (test22.lua)

Python 3 – none yet

k. **(5 Mar Python)** External DSL using Parser/Builder (no corresponding example in Fowler book chapter)

Lua (2013)

- Note: This program requires installation of a Lua LPEG library via luarocks. It must be compatible with whatever version of Lua is being used.
- builder module (builderLPEG1.lua)
- dsl script (rulesLPEG1.dsl)
- test driver (testLPEG1.lua)

Python 3 (2018)

- Note: This program requires installation of the Python 3 package Parsita, a parser combinator library similar to Scala's
- builder module (builderParsita1.py)
- dsl script (rulesParsita1.dsl)
- test driver (testParsita1.py)

14. **(5 Mar) Announce homework Assignment #3**

15. **(not discussed, but an alternative context for Assignment #3)** Use Sandwich DSL Case Study

    a. Notes

    - Haskell and Scala versions are similar expansions of the original Lua version
    - Haskell references below
    - Scala references below
    - Lua references below

    b. Haskell version (2014, 2017) using algebraic data types

    - Sandwich DSL case study (Haskell)
    - Haskell source

    c. Scala version (2016) using case classes

    - Sandwich DSL case study (Scala)
    - Scala source

    d. Lua version (2013) implements slightly different problem

    - Problem and solution description
    - Semantic Model module
    - Builder module implementing DSL applying Function Sequence pattern
    - Test driver

16. **(9 Mar)** No formal class. Work on new Assignment #3. Study the following handouts for discussion after Spring Break

    a. **(assigned first week of semester)** James Coplien, Daniel Hoffman, and David Weiss. Commonality and Variability in Software Engineering, *IEEE Software*, Vol. 15, No. 6, November 1998.

    b. Paper "A Little Language for Surveys: Constructing an Internal DSL in Ruby" BELOW

    c. Conrad Barski. Building a Text Game Engine, Chapter 5, In *Land of Lisp: Learn to Program in Lisp, One Game at a Time*, pp. 69-84, No Starch Press, 2011.

    The Common Lisp example in this chapter is similar to the classic Adventure game; the underlying data structure is a labeled digraph.

    There is m<ore information about both labeled digraphs and the Wizard's adventure game BELOW.

17. **(12-16 Mar) Enjoy Spring Break!**

Analysis and Design of DSLs

18. **(19-21, 2-6 Apr) Resume direct discussion of Domain Specific Languages notes**

19. **(21-26 Mar) Discuss commonality and variability analysis** paper:

    James Coplien, Daniel Hoffman, and David Weiss. Commonality and Variability in Software Engineering, *IEEE Software*, Vol. 15, No. 6, November 1998.

20. **(dates below) Examine Cunningham's Survey DSL case study**, in particular the analysis and DSL design

    a. **(28 Mar; 2 Apr) Discuss paper:** H. Conrad Cunningham. A Little Language for Surveys: Constructing an Internal DSL in Ruby, In *Proceedings of the ACM SouthEast Conference*, 6 pages, March 2008.

    - manuscript
    - presentation
    - improved paper introduction section

    b. **(2 Apr briefly)** Ruby source code (2006, 2008)

    - README file
    - Ruby source code SurveyLanguage.rb
    - test internal DSL input file
    - test internal DSL input file with errors

21. **(dates below) Study object-oriented software development**

    a. (for reference) Study domain analysis

    - (not updated) Instructor's slides on Domain Modeling (Powerpoint), for course registration system example
    - Alistair Cockburn. Introduction, Chapter 1 in *Writing Effective Use Cases*, Addison-Wesley, 2001.
    - Doug Rosenberg with Kendall Scott. Domain Modeling, Chapter 2 in *Use Case Driven Object Modeling with UML*, Addison Wesley 1999.
    - Doug Rosenberg with Kendall Scott. Use Case Modeling, Chapter 3 in *Use Case Driven Object Modeling with UML*, Addison Wesley 1999.

    b. **(6-13 Apr)** Instructor's notes on Object-Oriented Software Development

c. **(6-11 Apr)** Instructor's notes on Programming Paradigms – a draft "chapter" from the evolving "textbook" *Exploring Languages using Interpreters and Functional Programming*

- Object-oriented
- Prototype-based

d. **(9 Apr)** Discuss type systems terminology from Python 3 Metaprogramming notes

- Type system concepts
- Python type system

e. (for reference) Instructor's notes on the Kinds of Polymorphism

22. **(13 Apr) Distribute Assignment #4 (Final Project)**

23. **(dates below) Continue study of object-oriented software development**

a. **(16-18 Apr)** Instructor's slides on Using CRC Cards (Class-Responsibility-Collaboration) (HTML)

- (handout, for reference) Kent Beck and Ward Cunningham. A Laboratory for Teaching Object-Oriented Thinking, In *Proceedings of the OOPSLA'89 Conference*, ACM, 1989.
- (for reference) Paul Gestwicki. CRC Card Analysis (YouTube), Ball State University, February 2016.

b. (for reference) Ralph E. Johnson and Brian Foote. Designing Reusable Classes, *Journal of Object-Oriented Programming*, Vol. 1, No. 2, pages 22-35, June/July 1988.

c. (for reference) A simple example of the open-closed principle

d. (for reference) SOLID Design Principles, a look at the core SOLID principles through the lens of modularity.

e. (for reference) "Uncle" Bob Martin video on "SOLID Principles of Object-Oriented and Agile Design"

24. (for reference, slides need additional updating) Examine object-oriented programming

a. Understanding Inheritance, based on Timothy Budd's *Understanding Object-Oriented Programming with Java*, Chapter 8: [Powerpoint]

b. Software Reuse, based on Timothy Budd's *Understanding Object-Oriented Programming with Java*, Chapter 10: [Powerpoint]

- Scala translation of Frog dynamic composition example

c. Replacement and Refinement, loosely based on Timothy Budd's *An Introduction to Object-Oriented Programming*, Third Edition, Section 16.2: [Powerpoint]

d. Implications of Inheritance, based on Timothy Budd's *Understanding Object-Oriented Programming with Java*, Chapter 11: [Powerpoint]

e. Multiple Inheritance, based on Timothy Budd's *An Introduction to Object-Oriented Programming*, Third Edition, Chapter 13: [Powerpoint]

- Scala Modified Philosophical Frog example from Odersky et a
- Scala Modified Stackable traits example (IntQueue) from Odersky et al

f. Polymorphism, based on Timothy Budd's *Understanding Object-Oriented Programming with Java*, Chapter 12: [Powerpoint]

g. Second Look at Classes, loosely based on Timothy Budd's *An Introduction to Object-Oriented Programming*, Chapter 25 on Reflection and Introspection: [Powerpoint]

h. Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism, *ACM Computing Surveys*, Vol. 17, No. 4, pp. 471-523, 1985.

25. **(20-25 Apr) Do object-oriented analysis and design in class for a "choose your own adventure" game**

a. Reference: Conrad Barski. Building a Text Game Engine, Chapter 5, In *Land of Lisp: Learn to Program in Lisp, One Game at a Time*, pp. 69-84, No Starch Press, 2011.

The Common Lisp example in this chapter is similar to the classic Adventure game; the underlying data structure is a labeled digraph.

There is more information about both labeled digraphs and the Wizard's adventure game BELOW,

b. Adventure Game Analysis

Python 3 Metaprogramming

26. **(25-30 Apr, 2-4 May) Explore Python 3 Reflexive Metaprogramming**

a. Introduction [HTML] [PDF]

b. Basic Features Supporting Metaprogramming [HTML] [PDF]

c. Decorators and Metaclasses (for Debugging Case Study) [HTML] [PDF]]

Not Discussed Spring 2018 (For Reference)

Ruby DSLs

27. (for reference) Background on writing Ruby DSLs
    a. Jim Freeze. Creating DSLs with Ruby, Artima Developer, March 2006.
    b. Jamis Buck. Writing Domain Specific Languages, *The Bucklog* (blog), 20 April 2006.
    c. Rake (Ruby Make), a software management and build tool written as a Ruby internal DSL
28. (for reference) Discuss Fowler's DSL Reader framework
    a. Background:
        - Martin Fowler. Language Workbenches: The Killer-App for Domain Specific Languages? June 2005.
        - Martin Fowler. Generating Code for DSLs, June 2005.

    Ruby DSLs (2006)

    b. Ruby shared modules
        - DSL Reader Framework module (ReaderFramework.rb)
        - DSL Reader Utilities mix-in module (ReaderUtilities.rb)
        - Data input file (fowlerdata.txt)
        - Text DSL description (dslinput.txt)
        - XML DSL description (dslinput.xml)
    c. Ruby direct configuration and testing of Reader
        - BuilderDirect.rb
    d. Ruby single-pass external text DSL
        - TextSinglePass.rb
    e. Ruby two-pass external XML DSL
        - TwoPass.rb
        - class BuilderExternal source code generated by TwoPass.rb
    f. Ruby internal DSL
        - RubyDSL.rb

Language Processing

29. (for reference) Study Parsing Expression Grammars (PEGs)

a. Wikipedia entry on Parsing Expression Grammar

b. Bryan Ford. Packrat Parsing: Simple, Powerful, Lazy, Linear Time, Functional Pearl. In *Proceedings of the seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02)*, ACM, pp, 36-47, 2002.

c. Bryan Ford. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation, *ACM SIGPLAN Notices*, Vol. 39, No. 1, pp. 111-122, 2004. [slides]

30. (for reference) Arithmetic expression tree program skeletons

    a. Background: These are expanded from the example in Notes on Scala for Java Programmers.

    Scala versions (2008-16)

    b. Scala recursive function version using case classes

    c. Scala traditional object-oriented version

    Haskell version (2017)

    d. Haskell Expression Tree Calculator case study [source]

    Lua versions (2013-16)

    e. Lua recursive function versions (2013, 2014)

    - Lua Recursive Functions with Record Representation
    - Lua Recursive Functions with List Representation
    - Lua Evaluation Function Table with List Representation

    f. Lua object-oriented versions (2013, 2016)

    - Lua Prototype Object-Based
    - Lua Object-Oriented with Inheritance

    g. Lua LPEG parsers (2013)

    - Require installation of compatible LPEG library
    - Parser with captures
    - Parser with semantic actions (changed `unpack(t)` to `table.unpack(t)` for Lua 5.2 and 5.3)

Exploring Languages using Interpreters

Haskell

31. (for reference, from 2017 CSci 450) Explore the Expression Language Syntax and Semantics (Chapter 10)

    a. Supplemental slides:

- Expression Language Syntax and Semantics

b. Code in work:

- Module dependency graph
- Values module
- Abstract Syntax module
- Environments module
- Evaluator module
- Expression Language parsing and REPL modules linked to Chapter 11 below
- Process AST – simplification and derivative code skeleton

32. (for reference, from 2017 CSci 450) Explore the Expression Language Parsing (Chapter 11)

    a. Supplemental slides:

    - Expression Language Parsing

    b. Code in work:

    - Module dependency graph
    - Values module
    - Abstract Syntax module
    - Environments module
    - Evaluator module
    - Lexical analyzer module
    - Recursive descent parser for infix language (simple expressions)
    - Recursive descent parser for prefix language
    - Prefix REPL module
    - Infix REPL module
    - Process AST
    - Parser Combinators module

33. (for reference, from 2017 CSci 450) Explore the Expression Language Compilation (Chapter 12)

    a. Code in work:

    - Stack Virtual Machine?

34. (for reference, from 2017 CSci 450) Explore the Imperative Core Language: No notes or slide yet, but the code follows

    a. Modules

    - Values module
    - Abstract Syntax module
    - Environments module
    - Lexical analyzer module
    - Recursive descent parser module

- Evaluator module
- REPL module

Lua

35. (for reference, from 2016 CSci 450) Examine modularized Expression Language 1 interpreter

    a. Modules

    - Module Usage Diagram
    - Utilities module
    - Parser module [Infix parser] [Prefix parser]
    - Abstract Syntax module
    - Environment module
    - Values module
    - Evaluator module
    - REPL module

    b. Scripts to execute and test:

    - Evaluator Test script
    - Infix Parser Test script
    - Prefix Parser Test script
    - Run infix interpreter script
    - Run prefix interpreter script

36. (for reference, from 2016 CSci 450) Imperative Core Language

    a. Modules

    - Module Usage Diagram
    - Utilities module
    - Parser module
    - Abstract Syntax module
    - Environment module
    - Values module
    - Evaluator module
    - REPL module

    b. Run and test scripts

    - Evaluator Test script
    - Prefix Parser Test script
    - Run prefix interpreter script

Kamin Interpreters

37. (for reference, not updated) Kamin Interpreters in Lua Toolset (KILT)

a. Kamin-Budd Interpreters

   Samuel N. Kamin. *Programming Languages: An Interpreter-Based Approach*, Addison-Wesley, 1990.

   Note: I implemented the interpreter prototypes below in Lua 5.1 in 2013 for the Fall offering of CSci 658 and for a potential future offering of CSci 450. Some aspects my not work with Lua 5.2 or 5.3.

   I subsequently reimplemented the first with improved modularization in Lua in 2016 and then reimplemented it again in 2017 in Haskell.

b. Language/Interpreter-independent modules

   - [REPL Module (repl.lua)] (<KILT/repl.lua>)

   - Environment Module (environment.lua)
   - Function Table Module (funtab.lua)
   - Utilities Module (utilities.lua)
   - Opcodes Factory Module (opcodes.lua)
   - Values Factory Module (values.lua)
   - Parser Factory Module (parser.lua)
   - [Evaluator Factory Module (evaluator.lua)] (<KILT/evaluator.lua>)

c. Kamin Chapter 1 Core language interpreter

   - Core Interpreter (Core.lua)
   - Core Opcodes (opcodes_core.lua)
   - Core Values Module (values_core.lua)
   - Core Parser Module (parser_core.lua)
   - Core Evaluator Module (evaluator_core.lua)

d. Kamin Chapter 2 Lisp language interpreter

   - Lisp Interpreter (Lisp.lua)
   - Lisp Opcodes (opcodes_lisp.lua)
   - Lisp Values Module (values_lisp.lua)
   - Lisp Parser Module (parser_lisp.lua)
   - Lisp Evaluator Module (evaluator_lisp.lua)
   - A few Lisp examples

e. Kamin Chapter 4 Scheme language interpreter

   - Scheme Interpreter (Scheme.lua)
   - Scheme Opcodes (opcodes_scheme.lua)
   - Scheme Values Module (values_scheme.lua)
   - Scheme Parser Module (parser_scheme.lua)
   - Scheme Evaluator Module (evaluator_scheme.lua)
   - A few Scheme examples

Modules and Frameworks

38. (for reference) Simple, silly Employee hierarchy example

    a. Scala (2008, 2010)

    b. Ruby (2006)

39. (for reference) Examine a natural number arithmetic package

    This case study has implementations in five different languages (with some slight differences among the examples). So it can be used to compare implementations in different languages.

    a. Background on Peano arithmetic

        - Peano Axioms, Wikipedia article
        - Peano's Axioms, Wolfram MathWorld article

    b. Lua version (2013)

    c. Elixir version (2015)

        - Nat module
        - test module
        - test script

    d. Scala versions (2012, 2016)

        - Functional object-oriented style with ordinary classes
        - Functional object-oriented style with case classes
        - Functional module style with case classes

    e. Ruby version (2006)

    f. Java version (2004, 2016), simpler, no generics

        - abstract base class Nat
        - subclass Zero
        - subclass Succ
        - subclass Err
        - TestNat main program

40. (for reference) Abstract data types and modular design

    a. Background: Abstract data types

        - Nell Dale and Henry Walker. Abstract specification techniques, Chapter 1, In *Abstract Data Types: Specifications, Implementations, and Applications*, pp. 1-34,D. C. Heath, 1996.
        - H. Conrad Cunningham, Yi Liu, and Jingyi Wang. Designing a flexible framework for a table abstraction, Chapter 13 in Y. Chan, J. Talburt, and T. Talley, editors, *Data Engineering: Mining,*

Information, and Intelligence, pp. 279-314, Springer, 2010. [manuscript] [slides]

b. Modular Design

c. Data Abstraction: [slides]

d. Abstraction – draft "chapter" from evolving "textbook" Exploring Languages with Interpreters and Functional Programming (some overlap with Data Abstraction document)

e. Classic papers by David L. Parnas and associates:

- Kathryn Heninger Britton, R. Alan Parker, and David L. Parnas. A Procedure for Designing Abstract Interfaces for Device Interface Modules, In *Proceedings of the 5th International Conference on Software Engineering*, pp. 195-204, March 1981.
- David L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules, *Communications of the ACM*, Vol. 15, No. 12, pp. 1053-1058, 1972.
- David L. Parnas. On the Design and Development of Program Families, *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 1, pp. 1-9, March 1976.
- David L. Parnas. Designing Software for Ease of Extension and Contraction, *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 1, pp. 128-138, March 1979.
- David L. Parnas, P. C. Clements, and D. M. Weiss. The Modular Structure of Complex Systems, *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 3, pp. 259-266, March 1985.

f. William R. Cook. On Understanding Data Abstraction Revisited. In *Proceedings of OOPSLA*, October 2009. [local]

41. (for reference) Explore consequences of software architectural mismatch.

    Paper: David Garlan, Robert Allen, and John Ockerbloom. Architectural Mismatch: Why Reuse is So Hard, *IEEE Software*, Vol. 12, No. 6, November 1995.

    Instructor's notes on Architectural Mismatch

42. (for reference) Examine the CookieJar ADT case study (in Scala)

    a. Cookie Jar ADT Problem Description

    b. Specification concepts and notation – from Candy Bowl ADT description

    c. Immutable CookieJar ADT Implementation in Scala (`ICookieJar`) – uses method chaining functional style with immutable objects

    - ADT specification as Scala trait
    - List version

- HashMap version
- List of tuples version
- Blackbox test script

d. Mutable CookieJar ADT Implementation in Scala (`CookieJar`) – uses object-oriented style with mutable state

- ADT specification as Scala trait
- [List version ArrayBuffer version
- HashMap version
- List of tuples version
- Array version
- Blackbox test script

e. Similar specification and Ruby program

43. (for reference) Examine Carrie's Candy Bowl ADT (abstract data type) case study

a. Lua

- ADT Semantics
- Hashed version
- Unsorted List version
- Test driver

b. Scala

- CandyBowl trait
- CandyBowlList class

44. (for reference) Examine the Labeled Digraph case study

a. Background:

- Nell Dale and Henry Walker. "Directed Graphs or Digraphs," Chapter 10, In *Abstract Data Types: Specifications, Implementations, and Applications*, pp. 439-469, D. C. Heath, 1996.
- Conrad Barski. "Building a Text Game Engine,", Chapter 5, In *Land of Lisp: Learn to Program in Lisp, One Game at a Time*, pp. 69-84, No Starch Press, 2011.

  The Common Lisp example in this chapter is similar to the classic Adventure game; the underlying data structure is a labeled digraph.

b. Haskell solutions (2015)

- Labelled Digraph Abstract Data Type
- List representation for vertices and edges: [module] [test module]
- Map representation for graph: [module] [test module]

c. Elixir solutions (2015)

- Tuple and list representation for graph: [module] [test module]

d. Scala solutions (2016)

- Abstract data type specification and Scala trait (interface): [ADT interface]
- List representation for vertices and edges: [class source] [test source]
- Map (HashMap) representation for graph: [class source] [test source]

e. Using the Elixir Digraph ADT module to build the Wizard's Adventure game (2015)

- Wizard's Adventure game, Version 1, adapted from Chapter 5, 6, and 17 of Conrad Barski's *Land of Lisp: Learn to Program in Lisp, One Game at a Time*, No Starch Press, 2011.
- Wizard's Adventure game, Version 2, that uses a higher order function to generate game actions and improved handling of the game state.

45. (for reference) Examine the Dice of Doom game.

a. Background: Conrad Barski. *Land of Lisp: Learn to Program in Lisp, One Game at a Time*, No Starch Press, 2011.

b. Dice of Doom, Version 1a, is a basic eagerly evaluated version (similar to that developed on pages 303-325 of *Land of Lisp*). This version supports either two human players or a human player and a simple, minimax search-based "AI" (artificial intelligence) opponent.

c. Dice of Doom, Version 1b, is an eager version above with an attempt at memoization of functions `neighbors` and `game_tree` using the Elixir `Agent` modules.

d. Dice of Doom, Version 2a, is a lazy version using Elixir `Stream` data structures (but without the memoization optimizations). This is based on version 1a above plus the discussion from pages 384-389 in chapter 18 of *Land of Lisp*. This version implements a limited depth minimax search, but it does not implement the artificial intelligence heuristics given in the last part of chapter 18.

46. (for reference) Explore Systematic generalization.

a. Hans A. Schmid. Systematic Framework Design by Generalization, *Communications of the ACM*, Vol. 40, No. 10, pp. 48-51, October 1997.

b. Hans A. Schmid. Creating Applications From Components: A Manufacturing Framework Design, *IEEE Software*, Vol. 13, No. 6, November 1997.

c. Hans A. Schmid. Framework Design by Systematic Generalization, Chapter 15 in M. E. Fayad and R. E. Johnson, editors, *Domain-Specific Application Frameworks*, pp. 353-378, Wiley, 2000.

d. Hans A. Schmid. OSEFA: Framework for Manufacturing, Chapter 4 in M. E. Fayad, D. C. Schmidt, and R. E. Johnson, editors, *Building Application Frameworks: Object-Oriented Foundations of Framework Design*, pp. 43-65, Wiley, 1999.

47. (for reference) Mark Ardis, Nigel Daley, Daniel Hoffman, Harvey Siy, and David Weiss. Software Product Lines: A Case Study, *Software Practice and Experience*, Vol. 30, No. 7, pp. 825-847, 2000.

48. (for reference) Frameworks, based on Timothy Budd's *An Introduction to Object-Oriented Programming*, Third Edition, Chapter 21.

    a. Simple Sorting Framework in Scala

        - Low-level concrete Employee sorting
        - Insertion Sorting framework
        - Employee Sorting application of framework
        - Test code for Employee Sorting application of framework

    b. Ice Cream Store discrete event simulation

        - Simulation framework
        - Ice Cream Store application

49. (for reference) Scala Divide-and-Conquer Framework, similar to the Java framework in the paper:

    H. C. Cunningham, Y. Liu, and C. Zhang. Using classic problems to teach Java framework design, *Science of Computer Programming*, Special Issue on Principles and Practice of Programming in Java (PPPJ 2004), Vol. 59, No. 1-2, pp. 147-169, January 2006. doi: 10.10.16/j.scico.2005.07.009. [manuscript]

    Note: The above paper was not discussed in class, but the Scala versions of the Divide-and-Conquer (immediately below) and Binary Tree Traversal (farther down on this page) frameworks were discussed.

    - Template-based Divide-and-Conquer Framework (DivConqTemplate)
    - Strategy-based Divide-and-Conquer Framework (DivConqStrategy)
    - Traits for Problem and Solution descriptions for both frameworks (DivConqProblemSolution)
    - Application of Template-based framework to QuickSort (QuickSort-TemplateApp)
    - Application of Strategy-based framework to QuickSort (QuickSort-StrategyApp)
    - Descriptor for QuickSort state for both QuickSort applications (Quick-SortDesc)

50. (for reference) Binary Tree Visitor Scala Framework

    a. Background: H. C. Cunningham, Y. Liu, and C. Zhang. Using classic problems to teach Java framework design, *Science of Computer Programming*, Special Issue on Principles and Practice of Programming in Java (PPPJ 2004), Vol. 59, No. 1-2, pp. 147-169, January 2006. doi: 10.10.16/j.scico.2005.07.009.
    [manuscript]

    Scala versions (2008, 2010)

    b. Straightforward translation of the non-generic Java program to Scala

    - Top-level framework (BinTreeFramework)
    - Second-level Euler tour framework (EulerTourVisitor)
    - Second-level mapping visitor framework (MappingVisitor)
    - Second-level breadth-first framework (BreadthFirstVisitor)
    - Application of BinTree frameworks (BinTreeTest)

    c. Generic implementation of BinTree framework in Scala.

    - Top-level framework (BinTreeFramework)
    - Second-level Euler tour framework (EulerTourVisitor)
    - Second-level breadth-first framework (BreadthFirstVisitor)
    - Application of BinTree frameworks (BinTreeTest)

51. (for reference) Study the Movable and Named Objects case study (in Lua but based on a Haskell case study by Thompson)

    a. *Purpose*: The Named and Movable Objects case study explores use of inheritance hierarchies and multiple inheritance in Lua. I also extracted a "class support module" from the initial version of this code; this module was used in later case studies (e.g., Lair Configuration DSL).

    The "makeClass" function dynamically creates a "class" that has zero or more superclasses. It creates the class prototype object with an appropriate constructor/initialization function, appropriate default definitions of instance methods, and the needed settings of metaclasses and the `__index` metamethod to support the inheritance hierarchy.

    b. Background reading on object-oriented programming languages: Object-oriented subsection of the Fundamental Concepts of Programming Languages notes

    c. Background reading on case study: Section 14.6 of Simon Thompson. *Haskell: The Craft of Functional Programming*, Third Edition, Addison Wesley, 2011

    Note: We should readdress the design and implementation of this case study and the class support module. The latter may be too complex for our purposes in this course.

d. First Lua version (movable.lua)

e. Modularized Lua version with improved class support:

- class support module (class_support.lua)
- module using class-support (movable2.lua)
- test driver (movable2Test.lua)

f. Other older versions:<

- Haskell source
- Scala source (partial)

52. (for reference) Study the prototype-based programming example

Comments: Lua is essentially a *prototype-based* programming language. A Lua table has its own state, a unique identity, and an independent lifecyle. By assigning function closures to identifier-style keys, a table can also have its own operations. The method-call syntax enables these operations to refer to its associated table conveniently.

The Lua metatable and metamethod features enable a table to *delegate* some of its work conveniently to other tables. In particular, the `__index` metamethod enables accesses to data fields or operations not defined in one table to other tables.

The previous "object-oriented" Arithmetic Expression Tree and Movable & Named Objects examples use Lua's prototype-based features to implement classes and inheritance structures.

This module uses those features in a more straightforward prototype-based approach–by creating a "copy" of the prototype object. It provides two constructor functions, set up to use the method-call syntax.

Constructor method `Prototype:new` creates a new object as a *shallow copy* of the table parameter `mixin` with other references *delegated* to object `self`. A call `Prototype:new` creates a basic object that delegates to object `Prototype`. If `obj` has been created by this `new` method, then a call of `obj:new` creates a new object that delegates to `obj` rather than `Prototype`. (This is different from the way Lua classes are implemented.)

Constructor method `Prototype:clone` creates a new object that is a *shallow copy* of object `self` and then mixes in the fields and methods from `mixin`. It delegates accesses to any undefined fields to whatever object `self` does.

Concepts: Prototype object, delegation, shallow copy, mixin

- Prototype-based programming with delegation and cloning
- Test script

53. (for reference) Examine Rational Arithmetic case study (data abstraction)

Note: I have rewritten this 2016 case study to use Haskell and incorporated it into the draft Abstraction "chapter" of Exploring Languages with Interpreters and Functional Programming

a. Using Data Abstraction in Lua

   Example motivated by sections 2.1.1 and 2.1.2 from: Harold Abelson and Gerald J. Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs*, Second Edition, MIT Press, 1996

   SICP video lectures, Hal Abelson

   Concepts: designing modules with multiple implementations, data abstraction barrier ("building the wall"), canonical forms, using Lua modules, using higher order functions, using closures/thunks to encapsulate data

b. Rational arithmetic module – outer layer implementation of Rational Arithmetic abstraction

c. Rational number data representation using two-element arrays – primitive layer implementation 1
   [module] [test script]

d. Rational number data representation using array but deferring GCD – – primitive layer implementation 2
   [module] [test script]

e. Rational number data representation using closures – primitive layer implementation 3
   [module] [test script]

54. (for reference) Examine the complex number arithmetic modules in Lua

    Modules are repeated in each package in which they are used

    a. *Background* reading: *Structure and Interpretation of Computer Programs*, Second Edition, MIT Press, 1996, Section 2.4

       SICP video lectures, Hal Abelson

    b. Rectangular coordinates modules:
       [arithmetic] [rectangular representation] [utilities] [test driver]

    c. Polar coordinates modules:
       [arithmetic] [polar representation] [utilities] [test driver]

    d. Tagged data modules: [arithmetic] [data tagging] [utilities] [test driver]

    e. Data-directed programming modules:
       [arithmetic] [rectangular representation] [polar representation] [data tagging] [utilities] [test driver]

f. Object-oriented modules:
[arithmetic] [utilities] [test driver]

55. (for reference) Lua list module case study (Cell List)

Purpose: This Lua case study illustrates (i) functional programming principles, (ii) design and implementation methods for abstract data types and information hiding modules, and, (iii) Lua programming techniques (linked lists, stateless iterators, closures, metatables, etc.)

Background reading on Lua: Chapter 11 on data structures (pages 107-116) and Chapter 15 on modules (pages 151-161) of *Programming in Lua* (PiL), Third Edition

Caveats: (1) These modules (from 2013-14) internally layer the operations into primitive and non-primitive operations, but they do not separate the primitive operations into its own module. That can be done similarly to the Rational Arithmetic case study. (2) In the future, I plan to construct a more efficient implementation that uses array-style tables. The table-based versions likely also need to use weak tables to avoid memory leaks.

a. Cell-based list module:
[module source] [test driver]

b. Closure-and-table-based list module variant:
[module source] [test driver]

c. Function-based cell list module variant:
[module source] [test driver]

d. Lazy list module variant using C preprocessor (cpp -P):
[module source] [source after cpp]
[test driver] [driver after cpp] [sh script]

e. Lazy list module variant using Lua Macro 2.5:
[macro definitions] [module macro source] [source after luam -o]
[macro test driver] [driver after luam -o] [sh script]

56. (for reference) Examine functions adapted from SICP

a. Background reading: Chapter 1 of the classic textbook SICP – Harold Abelson and Gerald J. Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs*, Second Edition, MIT Press, 1996:
[book site at MIT Press] [HTML] [SICP ebook site

b. First-order functions in Scala

- Square root (Newton's Method) with all public functions
- Square root (Newton's Method) with nested function definitions
- Factorial
- Fibonacci

- Exponentiation
- Greatest common divisor

c. Higher-order functions in Scala

- Summation (takes function arguments)
- Derivative (returns function result)

d. Lua versions

- Square root (Newton's Method)
- Factorial
- Fibonacci
- Exponentiation
- Greatest common divisor
- Summation
- Derivative

e. Also various Haskell, Elixir, Elm, and Scheme versions

57. (for reference) Examine Square Root case study (stepwise refinement)

    a. Using Stepwise Refinement in Lua

    Concepts: TBD (examine the notes to see what was covered)

    Example motivated by sections 1.1.7 and 1.1.8 from: Harold Abelson and Gerald J. Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs*, Second Edition, MIT Press, 1996

    SICP video lectures, Hal Abelson

    b. Square root

Programming Language Reference Materials

58. Free online programming language textbooks and tutorials

Haskell

59. Learn X in Y Minutes, Where X = Haskell
60. H. Conrad Cunningham. Introduction to Functional Programming Using Haskell, Computer and Information Science, University of Mississippi, 2016-2017. (I am writing this "textbook" primarily for CSci 450 and likely will use the title *Exploring Languages with Interpreters and Functional Programming* for the Fall 2018 draft.)

61. H. Conrad Cunningham. *Notes on Functional Programming with Haskell*, Computer and Information Science, University of Mississippi, 1994-2014. (I am revising this document and integrating it with other materials to produce the new "textbook" above *Exploring Languages with Interpreters and Functional Programming*. The chapters at the chapters at the end have not yet been integrated.)

62. Haskell language website

    a. *Haskell 2010 Language Report*
    b. *GHC User's Guide*

63. *Haskell Cheat Sheet*

64. School of Haskell online tutorials

65. Miron Lipovaca. *Learn You a Haskell for Great Good: A Beginner's Guide*, a free online tutorial at http://learnyouahaskell.com/. (It is also in print from No Starch Press, 2011.)

66. Kees Doets and Jan van Eijck. *The Haskell Road to Logic, Math and Programming*, March 2004. This book is also available in print, published by College Publications, 2004.

67. Paul Hudak. *The Haskell School of Music: From Signals to Symphonies*, Version 2.6, January 2014. This book is a recent rewrite of Hudak's *The Haskell School of Expression: Learning Functional Programming through Multimedia*, Cambridge University Press, 2000.

68. Simon Marlowe. *Parallel and Concurrent Programming in Haskell: Techniques for Multicore and Multithreaded Programming*,

    2013. This book is also available in print, published by O'Reilly Media, 2013.

69. Bryan O'Sullivan, Don Stewart, and John Goerzen. *Real World Haskell*,

    2008. This book is also available in print, published by O'Reilly Media in November 2008.

Lua

70. Learn X in Y Minutes, Where X = Lua

71. Definitive reference: Roberto Ierusalimshcy. *Programming in Lua*, Fourth Edition, Lua.org, Rio de Janiero, Brazil, 2016. (The First Edition of this book, covering Lua 5.0, is available online at https://www.lua.org/pil/contents.html.)

72. Instructor's Lua slides (from CSci 450, Fall 2016)

a. Background: The slides below are adapted, in part, from *Programming in Lua, Slides for Course* by Fabio Mascarenhas from the Federal University of Rio de Janeiro, Brazil. He taught a course, which used Lua 5.2, at Nankai University, P. R. China, in July 2013.
b. Introduction to Lua slides based, in part, on Mascarenhas slide sets 0-5
c. Advanced Lua Functions slides based, in part, on Mascarenhas slide set 6
d. Modules in Lua slides based, in part, on Mascarenhas slide set 9
e. Lua Metatables slides based, in part, on Mascarenhas slide set 10
f. Lua Objects slides based, in part, on Mascarenhas slide set 11

Python 3

73. Learn X in Y Minutes, Where X = Python3
74. Bernd Klein. Python Course

Ruby

75. Learn X in Y Minutes, Where X = Ruby
76. Chris Pine. Learn to Program
77. why the lucky stiff (Jonathan Gillette). why's (poignant) guide to Ruby

Scala

78. Learn X in Y Minutes, Where X = Scala
79. Definitive reference: Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*, Third Edition, Artima, 2016. The first edition (2008) is available online at http://www.artima.com/pins1ed/.
80. Martin Odersky. *Scala by Example*, EPFL, 2014. [local]
81. Instructor's Notes on Scala for Java Programmers
82. Paul Chiusano and Runar Bjarnason. *Functional Programming in Scala*, Manning, 2015.
    a. Instructor's Notes on Functional Data Structures (Chapter 3)
    b. Instructor's Notes on Error Handling without Exceptions (Chapter 4)
    c. Instructor's Notes on Strictness and Laziness (Chapter 5)