Schedule of Topics
CSC 312, “Programming Language Implementation”
Department of Computer Science
Grinnell College
August 30, 2019

August 30: How does anyone go about designing and implementing a programming language? How are programming language implementations structured? What special techniques do implementers need to know?

Reading: Friedman and Wand, Essentials of Programming Languages, chapter 1 (pages 1–30)

September 2: How do we formally specify a recursively defined data type, such as lists, natural numbers, binary trees, or expressions of the \( \lambda \)-calculus? What notations are used to describe the structures of such types?

Reading: Friedman and Wand, chapter 2 (pages 31–55)

September 4: How can abstract data types be implemented in a programming language such as Scheme or Java? How do such formal specifications guide and determine the structures of procedures that operate on values of abstract data types? How are such data types used in the implementation of programming languages? How can we use the `define-datatype` and `cases` constructions to implement abstract data types easily in Scheme?

Reading: Friedman and Wand, from the beginning of chapter 3 through section 3.1 (pages 57–59); the (utilities character-sources), (LET tokens), and (LET scanner) libraries (in /home/reseda/programming-languages/code/)

September 6: How do lexical analyzers (scanners) work? What kinds of “tokens” do they produce?

Reading: Friedman and Wand, from the beginning of section 3.2 through subsection 3.2.1 (page 60), with special attention to Figure 3.2; the (LET syntax-trees) and (LET parser) libraries

September 9: How do parsers work? How are the abstract syntax trees that they produce structured?

Reading: Friedman and Wand, subsections 3.2.2 through 3.2.8 (pages 61–74); the (LET expvals), (LET environments), and (LET interpreters) libraries

September 11: How are environments used in the evaluation of expressions in programming languages? Why are they needed at all? What kinds of information is stored in them? How do the components of our implementation of LET fit together? How do the definitions of the abstract data types we use determine the structures of the procedures that operate on them? How can the implementation of LET be extended to include features like those mentioned in Exercises 3.6 through 3.11 and 3.15 through 3.18?

September 13: (pause for breath)

Reading: Friedman and Wand, section 3.3 (pages 74–82)

September 16: How can we have procedures as first-class values in a programming language? How do we implement procedures and procedure calls?

Reading: Friedman and Wand, sections 3.4 and 3.5 (pages 82–91)

September 18: How can we have recursive (and mutually recursive) procedures in a programming language? How do we manage the environments that contain bindings for such procedures? How do we determine the scope of a binding in such environments?
Schedule of Topics — page 2

Reading: Friedman and Wand, sections 4.1 and 4.2 (pages 103–113)

September 20: How can we have persistent side effects, such as assignments, in a programming language? How do we model and manage storage locations whose contents can be modified by programs as they are executed?

Reading: Friedman and Wand, section 4.3 (pages 113–124)

September 23: In programming languages without explicit reference values, how can we distinguish occurrences of variables that name storage locations (“lvalues”) from occurrences of variables that signify the expressed values stored in those locations (“rvalues”)?

Reading: Friedman and Wand, section 4.4 (pages 124–130)

September 25: How do we model storage structures, such as pairs and arrays?

Reading: Friedman and Wand, section 4.5 (pages 130–138)

September 27: How can storage locations be passed to procedures that have persistent side effects on them? In what different ways can the operands in a procedure call, or the values of those operands, be communicated to the procedure that is invoked? What are the advantages and disadvantages of deferring the evaluation of an operand expression until its value is actually needed for a computation?

Reading: Friedman and Wand, from the beginning of chapter 5 through section 5.1 (pages 139–155); the (LETREC-CPI continuations) library

September 30: How can we represent the “control context” in which an expression is evaluated? What information is stored in such a control context structure?

Reading: The (LETREC-CPI interpreter) library

October 2: What are the advantages and disadvantages of a continuation-passing interpreter? How is continuation passing related to tail-call optimization in Scheme?

October 4: (pause for breath)

Reading: Friedman and Wand, sections 5.2 and 5.3 (pages 155–171)

October 7: How could we implement a continuation-passing interpreter in a programming language without higher-order procedures and without tail-call optimization?

Reading: Friedman and Wand, section 5.4 (pages 171–178)

October 9: How can we raise and handle exceptions in a programming language?

Reading: Friedman and Wand, from the beginning of chapter 7 through section 7.3 (pages 233–248)

October 11: How can we enable programmers to declare types for variables and parameters? How can we arrange for type-related errors to be detected and reported before program evaluation begins?

Reading: Friedman and Wand, from the beginning of chapter 9 through section 9.4 (pages 325–352)

October 14: How can we implement classes, objects, and inheritance in a programming language?

Reading: Friedman and Wand, sections 9.5 and 9.6 (pages 352–371)

October 16: How can type declarations, type checking, and abstract classes (interfaces) be integrated into a programming language that includes classes and inheritance?

October 18: What questions relating to programming language implementation remain to be answered?