COMP 411/511

Principles of Programming Languages

Spring 2023

Professor Robert "Corky" Cartwright, Duncan Hall 3104

Department of Computer Science, Rice University, Houston, Texas, USA

Room 1064, Duncan Hall, Monday, Wednesday, Friday, 11:00am–11:50am

For more information on the course staff including office hours, see course information.

Course Resources

- This GitHub classroom site at https://classroom.github.com/classrooms/36009305-rice-comp411-511-spring-2023. The main course web page is https://ricecomp411.github.io/Master-2023/. It is available as a PDF file at https://ricecomp411.github.io/Master-2023/MainPage.pdf
- This course uses Piazza for class discussion and questions about class material. The site is located at the following URL: https://piazza.com/rice/spring2023/comp411/home. The Comp 411/511 Piazza discussion board will be monitored by the course staff and will also appear on each student's Piazza home page. The Piazza website is designed to quickly provide you with course help from classmates, the TAs, and myself. Rather than emailing questions to the teaching staff, I encourage you to post your questions on Piazza. If you have any problems or feedback for the Piazza developers, email team@piazza.com.
- Browser-based reference Jam interpreter developed by Nick Vrvilo, a recent doctoral graduate who now works for TwoSigma in Houston.

- **Language Resources**

1. Java
    a. JDK Download
    b. API Reference
    c. DrJava Programming Environment
    d. Elements of Object-Oriented Program Design by Prof. Cartwright
2. Scheme (Racket)
    a. How to Design Programs
    b. DrRacket Programming Environment

- **References from the Web**

1. Krishamurthi, Shriram. *Programming Languages: Application and Interpretation*. This book is a descendant of lecture notes created by Shriram for a version of this course when Shriram was a teaching assistant over a decade ago.
2. Friedman, Wand, and Haynes, *Essentials of Programming Languages*, 2nd ed. (MIT Press, 2001)
   You can take a look at the following two chapters, which the authors prepared for the second edition, without buying the book:
    a. Parameter Passing (Rice JavaPLT PDF)
    b. Types and Type Inference (Rice JavaPLT PDF)
3. Evaluation rules for functional Scheme (Rice JavaPLT PDF)
4. References on evaluating Jam programs
5. Lecture Notes on Types I
6. Lecture Notes on Types II
7. Introduction to System F (Polymorphic Lambda-Calculus)
8. Scheme code from Class Lectures
9. The Essence of Compiling with Continuations by Flanagan et al.
10. Uniprocessor Garbage Collection Techniques by Paul Wilson
11. Garbage Collection [canonical textbook] by Jones and Lins
12. Space Efficient Conservative Garbage Collection by Hans Boehm (Rice JavaPLT PDF)
13. Hans Boehm's Conservative GC Webpage
14. JVM Performance Optimization. The first article in a five part series (with the remaining four parts linked from the first article) on JVM internals and how to write Java source code to use them efficiently.
15. Java Memory Model
16. Revised Thread Synchronization Policies in DrJava (doc) (pdf). Since DrJava is built using the Java Swing library, it must conform to the synchronization policies for Swing. Unfortunately, the official Swing documentation is sparse and misleading in places, so this document includes a discussion of the Swing synchronization policies.
17. Lesson: Concurrency in Swing. This lesson discusses concurrency as it applies to Swing programming. It assumes that you are already familiar with the content of the Concurrency lesson in the Essential Classes trail.
18. Java concurrency (multi-threading): Tutorial. A tutorial on expressing concurrent computation in Java using threads.

19. The Last Word in Swing Threads. An article on the perils of accessing Swing components from outside the event dispatch thread.
20. Why Functional Programming Matters (as analyzed in 1990).
21. Why Functional Programming Mattered (gazing at programming technology in 2017).
22. Old Course Website

Course Summary

COMP 411/511 is an introduction to the principles of programming languages. It focuses on:

- identifying the conceptual building blocks from which languages are assembled and
- specifying the semantics, including common type systems, of programming languages.

In the lecture materials, interpreters are written in concise functional notation using Scheme (Racket). In some cases, supplementary material showing how the same interpreters can be written in another language such as Scala (using explicit typing) are provided. The functional subset of Scala and the ML family of languages (notably OCaml) are very well-suited to writing purely functional interpreters, yet they also support the limited use of mutation (imperativity) as well. I personally am not particularly fond of the ML family of languages for writing larger software systems because they do not support inheritance and object-oriented design. In contrast, type-safe OO languages like Java and Kotlin are very good vehicles for *implementing* these interpreters including the disciplined use of imperative code to greatly improve program efficiency. On the other hand, mainstream OO language do not provide a good representation for explaining abstract concepts because they are too wordy and the OO structure obscures the simple algebraic structure of abstract syntax and structural recursion used in simple interpreters. For this reason, we will present sample code in lecture in mostly functional Scheme/Racket. If you have not seen Scheme/Racket before, you will need to learn the core constructs of this language family which is widely imitated in many domain specific languages (DSLs) embedded in applications, such as Emacs Lisp embedded in Gnu Emacs.

A secondary theme of this course is software engineering. All of the programming assignments in this course are conducted in Java using *test-driven development*, a major tenet of Extreme Programming and other “agile” protocols for production software development. In the past, we have also recommended *pair programming* but that is not currently feasible given remote teaching mandates, social distancing constraints, and reliance on GitHub Classroom which makes the administration of project teams very cumbersome for both the course staff and project teams. A compelling reason for using Java is that it is widely used for software development in industry and it supports low-level programming where appropriate. Java Virtual Machines are expensive to start (the Achilles heel of Java IMO) and Java applications are compiled "Just In Time" (JIT) which has visible costs early in program execution. On the other hand, well-written Java code that has been "warmed up" (run on sufficient inputs to force the JIT compilation of all of the compute intensive parts of the application) is surprisingly fast.

COMP 411/511 consists of three parts:

- The first part focuses on specifying the syntax and the semantics of programming languages. The former is introduced via simple parsing (translations) of program text into abstract syntax. More detailed aspects of parsing (such as lexical analysis and advanced parsing methods) are left to COMP 412, a course on compilers. In practice, most domain specific languages do not need a sophisticated parser; in fact, so-called "internal DSLs" rely on the parser and compiler of the host language. Scala, a putative successor to Java which compiles to Java byte code, is specifically designed to support the easy implementation of internal DSLs. In addition, "external DSLs" (which are implemented by interpreters akin to the interpreter projects assigned in this course) often rely on simple manually written parsers based on "recursive descent" which is the approach to writing parsers taught in this course. The other attractive option for parsing external DSLs is to use a parser generator like ANTLR which is based on a simple extension of recursive descent parsing.

  In the course, you will implement the semantics of a comprehensive collection of programming language constructs using mostly functional interpreters derived from simple reduction (textual rewriting) semantics for the constructs. The constructs include arithmetical and conditional expressions, lexical binding of variables, blocks, first-class functions, assignment and mutation, basic control constructs (loop exits, first-class continuations, simple threads), and dynamic dispatch.

- The second part illustrates how interpreters and reduction semantics can be used to analyze important behavioral properties of programming languages. Two examples are covered: type safety and memory safety. Type safety guarantees that programs respect syntactically defined type abstraction boundaries and never raise certain classes of error signals. Memory safety guarantees that programs release memory if it is provably useless for the remainder of the evaluation. The standard technique for ensuring memory safety is automatic storage management, which is typically implemented using reference counting or garbage collection. Automatic storage management (which is present in all high-level programming languages including Java, Python, JavaScript, and Swift) never deletes data objects that are reachable in subsequent program execution. Type safety typically depends on memory safety. We will study the formal type-checking process incorporated in the ML family of languages, which has been adopted in part by sophisticated OO languages like Scala and Swift.

- The third part shows how interpreters can systematically be transformed so that they use fewer and fewer language facilities, demonstrating how to implement very high-level languages at the machine instruction level. The key transformations are the explicit representation of closures as records and the conversion of program control flow to continuation-passing style. Using these transformations, a recursive interpreter can readily be re-written in a low-level language like C/C++ or even assembly language. These transformations can be applied both to interpreters and to arbitrary programs. If we apply these transformations to an interpreter, the result is a low-level interpreter that is easily implemented in machine code. The process of applying these transformations to arbitrary input programs yields a high-level description of program compilation, a process which is explored in more detail in COMP 412.

This course material enables students to analyze the semantics and pragmatics of the old, new, and future programming languages that they are likely to encounter in the workplace (e.g., C, C++, Java, JavaScript, Swift, C#, Python). It also enables students to design and implement efficient interpreters for new languages or DSLs embedded in software applications. Most importantly, it equips students to become master software developers because they will be able to define and implement whatever linguistic extensions that are appropriate for simplifying the construction of a particular software system.

My notes on object-oriented program design briefly describe the design patterns that I recommend using to express functional programming abstractions in Java. If you have little prior experience in writing functional programming code in Java, I highly recommend skimming them.

For students interested in operational (syntax-based) semantics, I recommend reading notes by Walid Taha (now at Halmstad University in Sweden and Facebook in US) on big-step versus small-step semantics. I strongly prefer small-step (*syntactic*) semantics, so it is the only form of operational semantics that we will use in this class, but big-step semantics is often used in the programming languages literature.

Differences between COMP 411 and COMP 511

COMP 511 formerly included all of the material from COMP 411 plus a small amount of supplemental material. This year, that distinction is being dropped. COMP 511 is identical in content to COMP 411.

More Course Information

Please take a look at the course information/policies page, which is referenced in the first few class lectures.

Course Schedule

#	Date	Day	Topic	Reference	Assignment
1	1/10	M	Information & Motivation		
2	1/12	W	Parsing	Web site describing Syntax Diagrams (For EBNF see Project 1) Pascal Syntax Diagrams Tutorial on CFGs Java Lang Specification	
3	1/14	F			Project 1, assigned 1/14
	1/17	M	*Martin Luther King Holiday*		

#	Date	Day	Topic	Reference	Assignment
4	1/19	W	The Scope of Variables	*PLAI*, ch. 3-4	
5	1/21	F	Syntactic Interpreters		Project 1 due 10am, 1/21 Project 2 assigned 1/21
6	1/24	M		Brief Review of Lectures 1-6 Notes on OO Program Design	
7	1/26	W	Meta Interpreters Meta Errors	*PLAI*, Ch. 4-6	
8	1/28	F			
9	1/31	M			
10	2/2	W	Data Domains for Recursive Definitions	Intro to Domain Theory The Why of Y	
11	2/4	F			
12	2/7	M	Recursive Definitions and Environments	Recursive Programs in First-Order Logic Types as Intervals Lambda Calculus Model Supplemental Material	Project 2 due 10am, 2/7 Project 3 assigned 2/7
13	2/9	W	Recursive Binding		
	2/11	F	*Spring Recess*		
14	2/14	M	Eliminating Binding (SD λ-abstraction)	*Essentials*, ch. 3.7, 3.9 Hand Evaluation Exercises Solution to Hand Evaluation Exercises	
15	2/16	W	Assignment and Mutability		
16	2/18	F			
	2/21	M			Project 3 due 10am, 2/21 Project 4 and XC Project 4xc, assigned 2/21
17	2/23	W	Run-time Environment Representation and Control	*Essentials*, ch. 7, 8 Powerpoint slides from Sebesta's *Concepts of ProgrammingLanguages*	
18	2/25	F			
	2/28	M			
19	3/2	W	Object-Oriented Languages	*Essentials*, ch. 5	
	3/4	F	Sample MT Exam Solutions to 1,2,4,6 Solution to 3 Solution to 5(i) Solution to 5(ii)	*Essentials*, ch. 3.8	
	3/7	M			Project 4 due 10am, 3/7 Project 5/XC Project 5xc, assigned 3/7
	3/8	Tu	*MT Examination 7-10pm*		
20	3/9	W	What Is a Type? Types and Safety Types and Datatype	*Essentials*, ch. 4 *Essentials*, ch. 6 Type Inference Guide	
21	3/11	F			XC Project 4xc due 10am, 3/11

#	Date	Day	Topic	Reference	Assignment
	3/14-3/18	M-F	*Spring Break — No Classes*		
22	3/21	M	Polymorphism Implicit Polymorphism Survey of Unification	Draft Racket program for inefficient unification	
22'23	3/23	W	Final Words on Types		
24	3/25	F	Meaning of Function Calls	*Essentials*, ch. 7-8	Project 6 assigned 3/25 Project 5, 5xc due 10am, 3/25
25	3/28	M	Continuation-Passing Style		
26	3/30	W	Explaining `letcc` and `error`		
27	4/1	F			
28	4/4	M		Dynamic Storage Allocation Survey	
29	4/6	W	Garbage Collection	Uniprocessor Garbage Collection Techniques	
	4/8	F			Project 6 due 11am, 4/8 Project 7 assigned 4/8
	4/11	M			
30	4/13	W			
	4/17	F	Sample Final Exam Sample Final Exam With Solutions		
	4/20	M			
	4/22	W			
	4/24	F			Project 7 due 11:59pm 4/24

Accommodations for Students with Special Needs

Students with disabilities are encouraged to contact me during the first two weeks of class regarding any special needs. Students with disabilities should also contact Disabled Student Services in the Ley Student Center and the Rice Disability Support Services.