Principles of Compiler Construction (עקרונות הקומפילציה)

Dr Mayer Goldberg

October 20, 2018

Contents

- Course number: 201-1-2061
- Mandatory for undergraduate CS and SE students
- Credits: 4.5
- Course site: `http://www.cs.bgu.ac.il/~comp191/`
- Prerequisites: *Principle of Programming Languages* (202-1-2051), *Automata & Formal Languages* (202-1-2011), *Architecture* (202-1-3041)

1 Course Objectives

- Gain additional insight into programming languages, building on what students have learned in the *Principles of Programming Languages* course.

- Understand the major components of the compiler: Syntactic analysis, semantic analysis, code generation, and the run-time environment. Gain hands-on experience in crafting these components.
- Learn about compiler optimizations: What compilers do to generate code that is faster, shorter, and performs better. Implement many of these optimizations, and see how they improve the code.
- Be able to apply information & skills learned in the compilers course to other areas in computer science where syntactic and semantic analysis, code generation, and translation are needed.

2 Homework Guidelines

- The programming assignments may be submitted singly or in pairs. You may not work in larger groups.
- All assignment will be graded using *ocaml*, *Chez Scheme*, *nasm*, and *gcc* on the Linux image on the departmental lab computers only.

3 Detailed Syllabus

3.1 Introduction to Compiler Construction

References: 1, 3, 4, 5

- The algebraic relationship between compilation & interpretation.
- Cross-compilation, boot strapping a compiler, de-compilation.
- The stages of the compiler: What work is done in each, what kinds of errors can and cannot be detected at each, the basic algorithms that are implemented at each stage.
- Dynamic vs statically-typed languages. Early binding vs late binding. The information available to the compiler for translation, error detection, and optimizations.

3.2 Scanning & Parsing Theory

References: 1, 2, 5

- Scanner: DFA, NDFA, NDFA with ϵ-transitions

- Parsing: Top-down, recursive descent parsers, parsing combinators, bottom-up parsers
- Hand-coding various parsers
- Using parser-generation tools in C & Java
- Macro expansion: Syntactic transformations, reduction to core forms in the language, variables, meta-variables and syntactic hygiene.

3.3 Programming Languagues

References: 2, 3

- Functional vs Imperative programming: How change & side-effects are understood & modeled in the functional view of programming.
- Parameter-passing mechanisms: Call-by-value, call-by-reference, call-by-sharing/object, call-by-name, call-by-need. Features of the various parameter-passing mechanisms, their motivation, history & implementation.
- Scope & its implementation: Dynamic scope (deep binding, shallow binding), lexical scope. Dynamic scope and the implementation of exception handling.
- The structure of the lexical environment, and the implications for data sharing & side effects.
- Object-oriented vs functional programming Languagues. The structure of the closure compared to that of the object. Mapping of lambda-expressions to objects. The virtual method table.
- Monads & monadic programming.

3.4 Continuation-Passing Style (CPS)

References: 2, 5

- CPS as a programming technique (multiple return values, multiple continuations, co-routines, implementation of threads.
- CPS as an approach to writing a compiler: CPS, defunctionalization of the continuation, stack machine.

- CPS as an intermediate language for the compiler: Optimizations that are simpler in CPS.

3.5 Semantic Analysis

References: 3, 4, 5

- Lexical addressing, *deBruijn* numbering
- Identification of tail calls
- Boxing, data indirection, and motion from the stack to the heap: A comparison between quasi-functional programming languages (Scheme, LISP) and object oriented programming languages (Java).

3.6 Code Generation

References: 1, 2, 3, 4

- Layout of Scheme objects in memory. Run-time type information. Comparison with the situation in object-oriented programming languages.
- An overview of the proof of correctness of the compiler, and how it is constructed along with the code generator.
- Optimization of tail calls
- Code generation to native x86 instructions for the various expressions in our language
- The primitive procedures & support code that are provided with the compiler

3.7 The Run-Time Environment

References: 2, 3, 4

- The top level: *n*-LISP – value cells, function cells, property cells, etc.
- Dynamic memory management:
  - Reference counting
  - Garbage collection: mark & sweep, stop & copy, generational garbage collection
- Namespaces, modules, and their implementation

3.8 Compiler Optimizations

References: 1, 2, 3, and notes

- The tail-recursion & tail-call optimizations
- Loop optimizations & transformations
- Array optimizations
- Strength reduction optimizations
- Dead-code removal, write-after-write optimizations
- Common Sub-expression Elimination, both as a high-level and low-level optimization
- Optimizations for super-pipelined and parallel architectures

4 Final Grade

Your final grade is computed as follows. Your grade is made up of two components:

1. The Exam component
2. The Project component

It is possible to accumulate extra credit (bonus points) in both components, for up to a total of 113 points in total. While your final grade will not exceed 100 points, you can use these bonus points to compensate for lower grades on various deliverables.

4.1 The exam component

This component is worth 60 points. You have two options to fulfill this component:

- Option A: You take the final exam, and your grade makes up to 60 points of your final grade.
- Option B: You take both the midterm and the final exam.
  - The midterm grade will count up to 15 points of your final grade.

  - The final exam will count up to 50 points out of your final grade.

  The total points with this option is 65 points of your final grade. This is our way of encouraging you to attend the midterm, which is otherwise voluntary.

If you are unable to attend the midterm for any reason, just pick option A.

4.2 The project component

This component is worth 40 points.

- The online quizzes count up to 5 points of your final grade
- The final project will count up to 35 points of your final grade
- You will have 3–4 homework assignments:
  - If you submit an assignment on time, you can get a bonus of 2 points
  - If you don't have time to submit a specific assignment, just skip it, and move on
  - Regardless of whether or not you submit the assignments in a timely manner, you will need to submit a final project, on time, in order to pass the course

The final project is large and time-consuming, and you are given enough time during which to complete it. Nevertheless, we would like to encourage you to start working on the assignment early on, in a paced and timely manner. Therefore, the first 3 milestones of the final project have been split up into the first 3 assignments: Working on each of these 3 assignments means you shall be working on your final project; No extra work is involved. So basically, by submitting the first 3 assignments on time, you shall be earning up to 6 bonus points towards your final grade. The fourth, and last assignment will involve implementing various optimizations, and will not be a part of your final project. If you submit that assignment on time, you can accumulate up to 2 more bonus points towards your final grade.

If you choose option B of the exam component and submit the homework assignments on time, you can accumulate up to a theoretical total of 111 or 113 points, depending on whether there is a fourth assignment. Your grade shall not exceed 100, but 11 or 13 bonus points can make up for lost points

on various deliverables. This is our way of encouraging you to take the midterm, start working early on, and work in a paced and timely manner.

The *mandatory deliverables* in this course are the *final project* and the *final exam*. This means that you cannot pass this course if you fail any of these components.

Curving aims at setting the grades on a common scale. Curving in this course is optional, and at the sole discretion of the instructors, and can be done either up or down. The curving function need not be constant or linear, but it will be monotonic over the relevant interval.

5 Academic Integrity

The code you submit must have been written by yourself or your partner. It cannot come from other course members, friends, past students, or the internet. If you are caught having submitted code of which you or your partner are not the authors, or if you make your code available to other students, we will file a complaint against you with the disciplinary board (ועדת משמעת).

6 References

1. **Textbook:** Modern Compiler Design, by *D. Grune, H. Bal, C. Jacobs, K. Langendoen*
2. LISP in Small Pieces, by *Christian Queinnec*
3. The Anatomy of LISP, by *John Allen*
4. The Structure & Interpretation of Computer Programs, by *Harold Abelson, et al.*
5. Essentials of Programming Languages, by *Daniel P. Friedman, Mitchell Wand, Christopher T. Haynes*