Lecture 1

CIS 341: COMPILERS
Administrivia

• Instructor: Steve Zdancewic
  Office hours: Tuesdays 4:00-5:00 & by appointment
  Levine 511

• TAs:
  – Dmitri Garbuzov
  – Richard Zhang
  – JJ Lee
  – Vivek Raj
    • Office hours: To be determined

• E-mail: cis341@seas.upenn.edu
• Web site: http://www.seas.upenn.edu/~cis341
• Piazza: http://piazza.com/upenn/spring2017/cis341
Why CIS 341?

• You will learn:
  – Practical applications of theory
  – Lexing/Parsing/Interpreters
  – How high-level languages are implemented in machine language
  – (A subset of) Intel x86 architecture
  – More about common compilation tools like GCC and LLVM
  – A deeper understanding of code
  – A little about programming language semantics & types
  – Functional programming in OCaml
  – How to manipulate complex data structures
  – How to be a better programmer

• Expect this to be a very challenging, implementation-oriented course.
  – Programming projects can take tens of hours per week…
The CIS341 Compiler

• Course projects
  – HW1: OCaml Programming
  – HW2: X86lite interpreter
  – HW3: LLVMlite compiler
  – HW4: Lexing, Parsing, simple compilation
  – HW5: Higher-level Features
  – HW6: Analysis and Optimizations I
  – HW7: Optimizations II

• Goal: build a complete compiler from a high-level, type-safe language to x86 assembly.

*HW 4 – 7 are undergoing a re-design this semester, so they’re a bit in flux.
Resources

• Course textbook: (recommended, not required)
  – *Modern compiler implementation in ML* (Appel)

• Additional compilers books:
    • a.k.a. “The Dragon Book”
  – *Advanced Compiler Design & Implementation* (Muchnick)

• About Ocaml:
  – *Real World Ocaml* (Minsky, Madhavapeddy, Hickey)
    • realworldocaml.org
  – *Introduction to Objective Caml* (Hickey)
Why OCaml?

• OCaml is a dialect of ML – “Meta Language”
  – It was designed to enable easy manipulation abstract syntax trees
  – Type-safe, mostly pure, functional language with support for polymorphic (generic) algebraic datatypes, modules, and mutable state
  – The OCaml compiler itself is well engineered
    • you can study its source!
  – It is the right tool for this job

• Forgot about OCaml after CIS120?
  – Next couple lectures will (re)introduce it
  – First two projects will help you get up to speed programming
  – See “Introduction to Objective Caml” by Jason Hickey
    • book available on the course web pages, referred to in HW1
HW1: Hellocaml

• Homework 1 is available on the course web site.
  – Individual project – no groups
  – Due: Thursday, 19 Jan. 2013 at 11:59pm
  – Topic: OCaml programming, an introduction

• OCaml head start on eniac:
  – Run “ocaml” from the command line to invoke the top-level loop
  – Run “ocamlbuild main.native” to run the compiler

• We recommend using:
  – Emacs/Vim + merlin
  – (less recommended: Eclipse with the OcaIDE plugin)

  – See the course web pages about the CIS341 tool chain to get started
Homework Policies

• Homework (except HW1) may be done individually or in pairs

• Late projects:
  – up to 24 hours late: 15 point penalty
  – up to 48 hours late: 30 point penalty
  – after 48 hours: not accepted

• Submission policy:
  – Projects that don’t compile will get no credit
  – Partial credit will be awarded according to the guidelines in the project description

• Academic integrity: don’t cheat
  – This course will abide by the University’s Code of Academic Integrity
  – “low level” and “high level” discussions across groups are fine
  – “mid level” discussions / code sharing are not permitted
  – General principle: When in doubt, ask!
Course Policies

Prerequisites: CIS121 and CIS240 (262 useful too!)
   – Significant programming experience
   – If HW1 is a struggle, this class might not be a good fit for you
     (HW1 is significantly simpler than the rest…)

Grading:
• 70% Projects: *Compiler*
   – Groups of 1 or 2 students
   – Implemented in OCaml

• 12% Midterm
• 18% Final exam

• Lecture attendance is crucial
• *No laptops (or other devices)!*
  – It’s too distracting for me and for others in the class.
What is a compiler?
What is a Compiler?

• A compiler is a program that translates from one programming language to another.
• Typically: *high-level source code* to *low-level machine code* (object code)
  – Not always: Source-to-source translators, Java bytecode compiler, GWT
    Java ⇒ Javascript
Historical Aside

- This is an old problem!
- Until the 1950’s: computers were programmed in assembly.
- 1951—1952: Grace Hopper developed the A-0 system for the UNIVAC I
  - She later contributed significantly to the design of COBOL
- 1957: the FORTRAN compiler was built at IBM
  - Team led by John Backus
- 1960’s: development of the first bootstrapping compiler for LISP
- 1970’s: language/compiler design blossomed

Today: thousands of languages (most little used)
  - Some better designed than others...

1980s: ML / LCF
1984: Standard ML
1987: Caml
1991: Caml Light
1995: Caml Special Light
1996: Objective Caml
• Optimized for human readability
  – Expressive: matches human ideas of grammar / syntax / meaning
  – Redundant: more information than needed to help catch errors
  – Abstract: exact computation possibly not fully determined by code
• Example C source:

```c
#include <stdio.h>

int factorial(int n) {
    int acc = 1;
    while (n > 0) {
        acc = acc * n;
        n = n - 1;
    }
    return acc;
}

int main(int argc, char *argv[]) {
    printf("factorial(6) = %d\n", factorial(6));
}
```
Low-level code

• Optimized for Hardware
  – Machine code hard for people to read
  – Redundancy, ambiguity reduced
  – Abstractions & information about intent is lost

• Assembly language
  – then machine language

• Figure at right shows (unoptimized) 32-bit code for the factorial function

```asm
_factorial:
## BB#0:
    pushl %ebp
    movl %esp, %ebp
    subl $8, %esp
    movl 8(%ebp), %eax
    movl %eax, -4(%ebp)
    movl $1, -8(%ebp)
LBB0_1:
    cmpl $0, -4(%ebp)
    jle LBB0_3
## BB#2:
    movl -8(%ebp), %eax
    imull -4(%ebp), %eax
    movl %eax, -8(%ebp)
    movl -4(%ebp), %eax
    subl $1, %eax
    movl %eax, -4(%ebp)
    jmp LBB0_1
LBB0_3:
    movl -8(%ebp), %eax
    addl $8, %esp
    popl %esp
    retl
```
How to translate?

• Source code – Machine code mismatch
• Some languages are farther from machine code than others:
  – Consider: C, C++, Java, Lisp, ML, Haskell, Ruby, Python, Javascript

• Goals of translation:
  – Source level expressiveness for the task
  – Best performance for the concrete computation
  – Reasonable translation efficiency (< O(n^3))
  – Maintainable code
  – Correctness!
Programming languages describe computation precisely…

– therefore, translation can be precisely described
– a compiler can be correct with respect to the source and target language semantics.

Correctness is important!

– Broken compilers generate broken code.
– Hard to debug source programs if the compiler is incorrect.
– Failure has dire consequences for development cost, security, etc.

This course: some techniques for building correct compilers

– Finding and Understanding Bugs in C Compilers,
  Yang et al. PLDI 2011

– There is much ongoing research about proving compilers correct.
  (Google for CompCert, Verified Software Toolchain, or Vellvm)
Idea: Translate in Steps

• Compile via a series of program representations

• Intermediate representations are optimized for program manipulation of various kinds:
  – Semantic analysis: type checking, error checking, etc.
  – Optimization: dead-code elimination, common subexpression elimination, function inlining, register allocation, etc.
  – Code generation: instruction selection

• Representations are more machine specific, less language specific as translation proceeds
(Simplified) Compiler Structure

Source Code
( Character stream)
if (b == 0) a = 0;

Lexical Analysis

Token Stream

Parsing

Abstract Syntax Tree

Intermediate Code Generation

Intermediate Code

Code Generation

Front End
(machine independent)

Middle End
(compiler dependent)

Back End
(machine dependent)

Assembly Code
CMP ECX, 0
SETBZ EAX
Typical Compiler Stages

- Lexing → token stream
- Parsing → abstract syntax
- Disambiguation → abstract syntax
- Semantic analysis → annotated abstract syntax
- Translation → intermediate code
- Control-flow analysis → control-flow graph
- Data-flow analysis → interference graph
- Register allocation → assembly
- Code emission

- Optimizations may be done at many of these stages
- Different source language features may require more/different stages
Compilation & Execution

Source code

Compiler

Assembly Code

Assembler

Object Code

Linker

Library code

Fully-resolved machine Code

Loader

Executable image

foo.c

gcc -S

foo.s

as

foo.o

ld

foo

(Usually: gcc -o foo foo.c)
Introduction to OCaml programming
A little background about ML
Interactive tour via the OCaml top-loop & Eclipse
Writing simple interpreters
ML’s History

• **1971**: Robin Milner starts the LCF Project at Stanford
  – “logic of computable functions”

• **1973**: At Edinburgh, Milner implemented his theorem prover and dubbed it “Meta Language” – ML

• **1984**: ML escaped into the wild and became “Standard ML”
  – SML ‘97 newest version of the standard
  – There is a whole family of SML compilers:
    • SML/NJ – developed at AT&T Bell Labs
    • MLton – whole program, optimizing compiler
    • Poly/ML
    • Moscow ML
    • ML Kit compiler
    • MLj – SML to Java bytecode compiler

• **ML 2000**: failed revised standardization

• **sML**: successor ML – discussed intermittently

• **2014**: sml-family.org + definition on github
OCaml’s History

• The Formel project at the Institut National de Recherche en Informatique et en Automatique (INRIA)
• 1987: Guy Cousineau re-implemented a variant of ML
  – Implementation targeted the “Categorical Abstract Machine” (CAM)
  – As a pun, “CAM-ML” became “CAML”
• 1991: Xavier Leroy and Damien Doligez wrote Caml-light
  – Compiled CAML to a virtual machine with simple bytecode (much faster!)
• 1996: Xavier Leroy, Jérôme Vouillon, and Didier Rémy
  – Add an object system to create OCaml
  – Add native code compilation
• Many updates, extensions, since…
• Microsoft’s F# language is a descendent of OCaml
• 2013: ocaml.org
OCaml Tools

- ocaml – the top-level interactive loop
- ocamlc – the bytecode compiler
- ocamlopt – the native code compiler
- ocamldep – the dependency analyzer
- ocamldoc – the documentation generator
- ocamlllex – the lexer generator
- ocamlyyacc – the parser generator
- menhir – a more modern parser generator
- ocamlbuild – a compilation manager
- utop – a more fully-featured interactive top-level
- opam – package manager
Distinguishing Characteristics

• Functional & (Mostly) “Pure”
  – Programs manipulate values rather than issue commands
  – Functions are first-class entities
  – Results of computation can be “named” using \texttt{let}
  – Has relatively few “side effects” (imperative updates to memory)

• Strongly & Statically typed
  – Compiler typechecks every expression of the program, issues errors if it can’t prove that the program is type safe
  – Good support for type inference & generic (polymorphic) types
  – Rich user-defined “algebraic data types” with pervasive use of \textit{pattern matching}
  – Very strong and flexible module system for constructing large projects
Most Important Features for CIS341

• Types:
  – int, bool, int32, int64, char, string, built-in lists, tuples, records, functions

• Concepts:
  – Pattern matching
  – Recursive functions over algebraic datatypes

• Libraries:
  – Int32, Int64, List, Printf, Format
How to represent programs as data structures.
How to write programs that process programs.

INTERPRETERS
Consider this implementation of factorial in a hypothetical programming language:

```plaintext
x = 6;
ANS = 1;
whileNZ (x) {
    ANS = ANS * x;
    x = x + -1;
}
```

We need to describe the constructs of this hypothetical language

- **Syntax**: which sequences of characters count as a legal “program”?
- **Semantics**: what is the meaning (behavior) of a legal “program”?
Grammar for a Simple Language

\[
<\text{exp}> ::= \\
\quad \mid \quad <\text{X}> \\
\quad \mid \quad <\text{exp}> + <\text{exp}> \\
\quad \mid \quad <\text{exp}> * <\text{exp}> \\
\quad \mid \quad <\text{exp}> < <\text{exp}> \\
\quad \mid \quad <\text{integer constant}> \\
\quad \mid \quad (<\text{exp}>)
\]

\[
<\text{cmd}> ::= \\
\quad \mid \quad \text{skip} \\
\quad \mid \quad <\text{X}> = <\text{exp}> \\
\quad \mid \quad \text{ifNZ} <\text{exp}> \{ <\text{cmd}> \} \text{ else } \{ <\text{cmd}> \} \\
\quad \mid \quad \text{whileNZ} <\text{exp}> \{ <\text{cmd}> \} \\
\quad \mid \quad <\text{cmd}>; <\text{cmd}>
\]

- Concrete syntax (grammar) for a simple imperative language
  - Written in “Backus-Naur form”
  - \(<\text{exp}>\) and \(<\text{cmd}>\) are nonterminals
  - ‘::=’, ‘|’, and \(<\ldots>\) symbols are part of the meta language
  - keywords, like ‘\text{skip}’ and ‘\text{ifNZ}’ and symbols, like ‘{‘ and ‘+’ are part of the object language

- Need to represent the abstract syntax (i.e. hide the irrelevant of the concrete syntax)
- Implement the operational semantics (i.e. define the behavior, or meaning, of the program)
OCaml Demo

simple.ml