Lecture 1

COMPILER DESIGN
**Administrivia**

- **Instructor:** Prof. Zhendong Su / Dr. Tobias Grosser  
  **Office:** CNB H 102
- **TAs:**  
  - Dr. Manuel Rigger (Lead TA)  
  - Dominik Winterer  
  - Additional TAs TBA
- **Web site:** [https://people.inf.ethz.ch/suz/teaching/252-0210.html](https://people.inf.ethz.ch/suz/teaching/252-0210.html)  
  **Moodle:** [https://moodle-app2.let.ethz.ch/course/view.php?id=11669](https://moodle-app2.let.ethz.ch/course/view.php?id=11669)  
  **E-mail for teaching staff:** TBA (see course web site later)
Why Study Compilers?

• You will learn
  – Practical applications of theory
  – Lexing / Parsing / Interpreters
  – How high-level languages are implemented in machine languages
  – (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 Compiler Project

- Course projects
  - HW1: Ocaml programming (due 01.10)
  - HW2: X86lite interpreter (due 15.10)
  - HW3: LLVMlite compiler (due 29.10)
  - HW4: Lexing, parsing, simple compilation (due 12.11)
  - HW5: Higher-level features (due 26.11)
  - HW6: Analysis and optimizations (due 10.12)

- Goal: Build a complete compiler from a high-level, type-safe language to x86 assembly
**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 of 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

• Haven’t learned OCaml?
  – Next couple lectures (& the first exercise session) will 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 page (also referred to in HW1)
HW1: Helloworld

- Homework 1 will be available on the course Moodle site
  - Individual project – no groups
  - Due: Tuesday, 1 Oct. at 23:59
  - Topic: OCaml programming, an introduction to interpreters

- OCaml head start
  - 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 OcamlDE plugin)

  - More information on the tool chain will be on course moodle or website
Homework Policies

• Homework (except HW1) should be done in pairs or individually
  – Please start forming teams
• 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
  – Submissions that don’t compile will receive no credit (sorry!)
  – Partial credit will be awarded following guidelines in project descriptions
• 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, please ask!
Course Policies

Prerequisites
- Significant programming experience
- Exposure to modern techniques for program construction
- Knowledge of some processor architectures at the assembly level
- If HW1 is a struggle, this class might not be a good fit for you
  (HW1 is significantly simpler than the rest of the assignments)

Grading:
- 50% Projects: *The Compiler*
  - Groups of 1 or 2 students
  - Implemented in OCaml
- 50% Final exam
- Lecture & exercise attendance is crucial
Lecture Schedule (tentative)

On course website: https://people.inf.ethz.ch/suz/teaching/252-0210.html

18.09  Introduction: Compilers, Interpreters, OCaml
19.09  OCaml Crash Course: Translating Simple to OCaml
25.09  X86lite
26.09  X86lite programming / C calling conventions
02.10  Intermediate Representations I
03.10  Intermediate Representations II
09.10  Intermediate Representations III / LLVM
10.10  Structured Data in the LLVM IR
16.10  Lexing: DFAs and ocamllex
17.10  Parsing I: Context Free Grammars
23.10  Parsing II: LL(k) parsing, LR(0) parsing
24.10  Parsing III: LR(1) Parsing and Menhir
30.10  First-class Functions I
31.10  First-class Functions II: Interpreters
06.11  Closure Conversion and Types I
07.11  Types II: Judgments and Derivations
13.11  Subtyping
14.11  OO: Dynamic Dispatch and Inheritance
20.11  Multiple Inheritance & Optimizations I
21.11  Optimizations II / Data Flow Analysis
27.11  Register Allocation
28.11  Data Flow Analysis II
04.12  Control Flow Analysis / SSA Revisited
05.12  Selected Topics: Garbage Collection
11.12  Selected Topics: Compiler Testing
12.12  Selected Topics: Compiler Verification
18.12  Selected Topics: MLIR
19.12  Course Summary
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 Notes

• 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: IBM built the FORTRAN compiler
  – 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

```assembly
_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 %ebp
  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!
Correct Compilation

• 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
  – Compiler Validation via Equivalence Modulo Inputs, Le et al. PLDI 2014

  – There is much ongoing research about proving compilers correct
    (Google for CompCert, Verified Software Toolchain, or Vellvm)
LLVM Bug #14972

```c
struct tiny { char c; char d; char e; };

void foo(struct tiny x) {
    if (x.c != 1) abort();
    if (x.e != 1) abort();
}

int main() {
    struct tiny s;
    s.c = 1; s.d = 1; s.e = 1;
    foo(s);
    return 0;
}
```

```bash
$ clang -m32 -O0 test.c ; ./a.out
$ clang -m32 -O1 test.c ; ./a.out
Aborted (core dumped)
```
Idea: Translate in Steps

• Compile via a series of program representations

• Intermediate representations (IRs) 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

Assembly Code
CMP ECX, 0
SETBZ EAX

Front End
(machine independent)

Middle End
(compiler dependent)

Back End
(machine dependent)
### Typical Compiler Stages

<table>
<thead>
<tr>
<th>Stage</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lexing</td>
<td>token stream</td>
</tr>
<tr>
<td>Parsing</td>
<td>abstract syntax</td>
</tr>
<tr>
<td>Disambiguation</td>
<td>abstract syntax</td>
</tr>
<tr>
<td>Semantic analysis</td>
<td>annotated abstract syntax</td>
</tr>
<tr>
<td>Translation</td>
<td>intermediate code</td>
</tr>
<tr>
<td>Control-flow analysis</td>
<td>control-flow graph</td>
</tr>
<tr>
<td>Data-flow analysis</td>
<td>interference graph</td>
</tr>
<tr>
<td>Register allocation</td>
<td>assembly</td>
</tr>
<tr>
<td>Code emission</td>
<td></td>
</tr>
</tbody>
</table>

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

- Assembly code is not the end of the story
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 & Emacs
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
- ocamllex – the lexer generator
- ocamlyacc – 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 (i.e., supports higher-order functions)
  – Results of computation can be “named” using `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 pattern matching
  – Very strong and flexible module system for constructing large projects
Most Important Features for this Class

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

• Concepts
  – Pattern matching
  – Recursive functions over algebraic (i.e. tree-structured) datatypes

• Libraries
  – Int32, Int64, List, Printf, Format
How to represent programs as data structures.
How to write programs that process programs.
Factorial: Everyone’s Favorite Function

• 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

\[
\begin{align*}
<exp> & ::= \\
& \mid <X> \\
& \mid <exp> + <exp> \\
& \mid <exp> * <exp> \\
& \mid <exp> < <exp> \\
& \mid <integer \ constant> \\
& \mid (<exp>)
\end{align*}
\]

\[
\begin{align*}
<cmd> & ::= \\
& \mid \text{skip} \\
& \mid <X> = <exp> \\
& \mid \text{ifNZ} <exp> \{ \langle cmd \rangle \} \text{ else } \{ \langle cmd \rangle \} \\
& \mid \text{whileNZ} <exp> \{ \langle cmd \rangle \} \\
& \mid \langle cmd \rangle ; \langle cmd \rangle
\end{align*}
\]

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