YSC4230: Programming Language Design and Implementation

Ilya Sergey

ilya.sergey@yale-nus.edu.sg

ilyasergey.net/YSC4230/

Interpreters and Compilers

https://www.youtube.com/watch?v=_C5AHaS1mOA

(From Episode 6 of the classic 1983 television series, Bits and Bytes)

Week 1: Introduction

Why PLDI?

- You will learn:
  - How programs we write end up being executed
  - Practical applications of Programming Language 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
  - How to better understand program code
  - A little about programming language semantics & types (*math* behind programs)
  - Advanced functional programming in OCaml (yay!)
  - How to manipulate complex data structures
  - How to be a better programmer

- Expect this to be a *very challenging*, implementation-oriented course (duh!)
  - Programming projects can take *tens* of hours per week…

Administrivia

- **Instructor:** Ilya Sergey
  Lectures (F2F): Wednesdays, 9am-12pm,
  Elm Common Lounge
  Office hours: Mondays, 3:30pm-5:00pm (please, email me upfront)
  #RC3-01-03E, Cendana
- **E-mail:** ilya.sergey@yale-nus.edu.sg
- **Web site:** https://ilyasergey.net/YSC4230
- **GitHub:** https://github.com/ysc4230

Please, email me your GitHub name to access the code!

Course Policies

Prerequisites: YSC1212 and YSC2229

- Significant programming experience
- Familiarity with data Structures
- HW1 will refresh your knowledge of OCaml

Grading:

- 85%: coding projects: *Compiler*
  - Groups of 2 students (except the 1st one)
  - Implemented in OCaml

- 10%: paper-based research project (individual)
  - Writing a short review on a state-of-the-art paper in PLDI

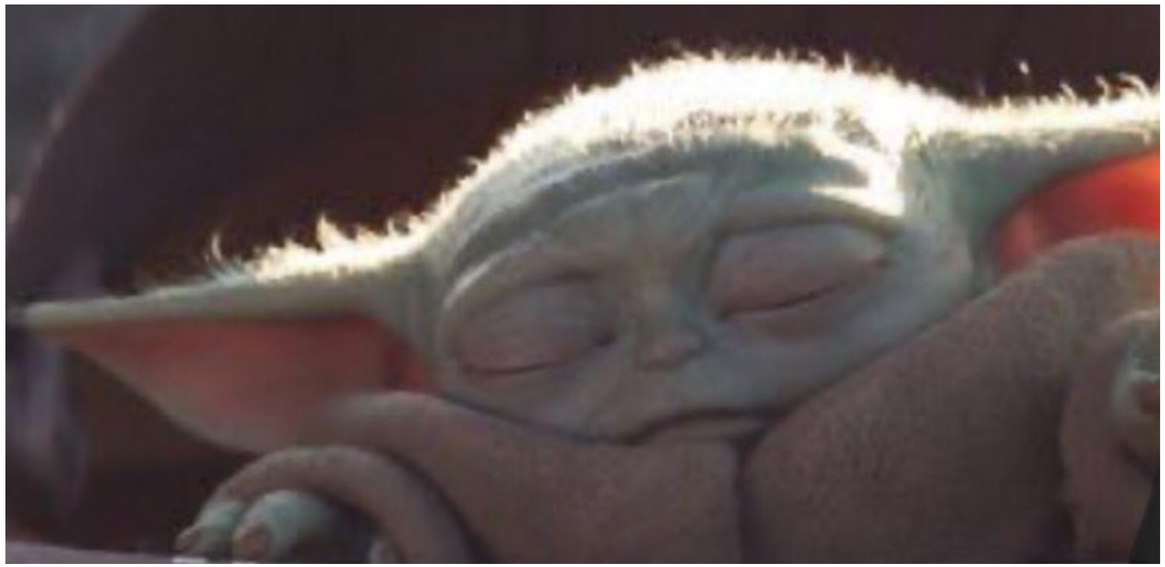
- Lecture attendance is crucial (5% of the final grade)
  - Active participation (asking questions, etc.) is encouraged

Homework Projects

- Six homework assignments (each graded out of 100 points)
  - HW1: OCaml Programming
  - HW2: X86lite interpreter
  - HW3: LLVMlite compiler
  - HW4: Lexing, Parsing, simple compilation
  - HW5: Higher-level Features
  - HW6: Analysis and Optimisations

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

General Advice



- Morning class: most of us are sleepy at 9am.
  Try to make class livelier by asking questions and participating in discussions!

Homework Policies

- Homework (except HW1) should be done *individually* or *in pairs*
- Late projects:
  - up to 24 hours late: 10 point penalty
  - up to 48 hours late: 20 point penalty
  - after 48 hours: not accepted (sorry)

- Submission policy:
  - Projects that **don't compile will get no credit**
  - Partial credit will be awarded according to the guidelines in the project description

- Fair work-split policy:
  - In group projects it is expected each member to contribute non-trivial amount of code (not comments, blank lines or trivial code permutations);
  - I will use GitHub contribution tracking for this; please, make sure your email is properly configure with GitHub so this accounting would work;
  - "Freeloaders" will be penalised at my discretion.

Academic Integrity

- "low level" and "high level" discussions across groups are fine
  - **"Low level"**: "how do you debug your LLVM output?", "what is the meaning of this x86 operation?"
  - **"High level"**: "What is a lattice in a data flow analysis?", drawing boxes on a whiteboard.

- "mid level" discussions / code sharing between teams are **not permitted**
  - **"Mid-level":** "how does your type checker implementation work on lambdas?"

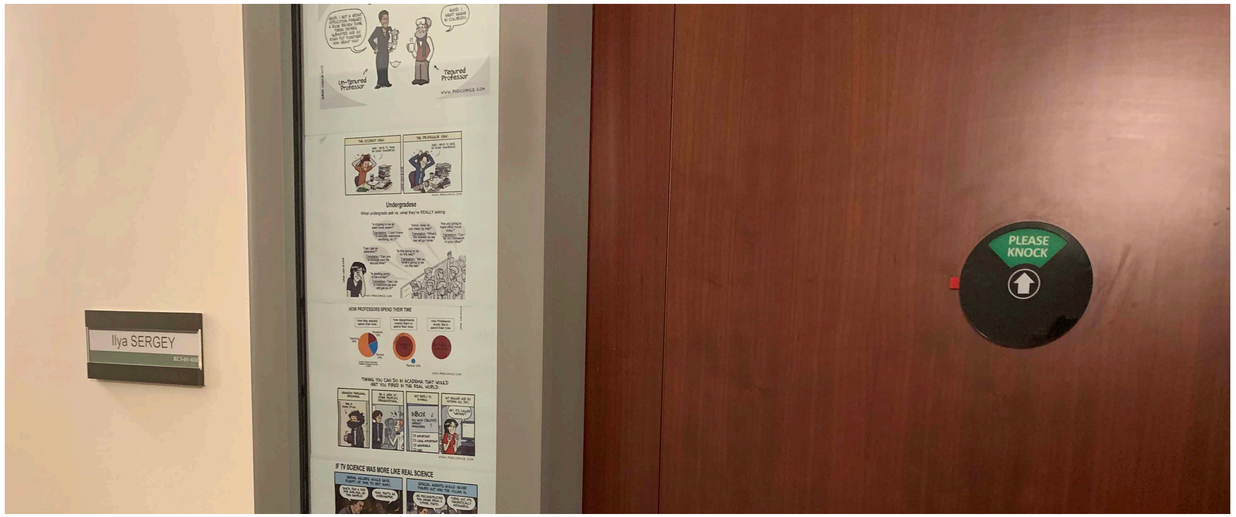
- Adopting/translating code parts from the internet is **not permitted** (I will know)

- General principle: *When in doubt, ask the instructor!*

- Penalties for cheating:
  - First strike: 0 points to the whole team for the homework
  - Second strike: F for the module, case is passed to the Academic Integrity Committee

Getting Help

- Office Hours (#RC3-01-03E, Cendana):
  Mondays 15:30-17:00 (preferred)
  Wednesdays 17:15-18:30
  Please, email me upfront!



- **E-mail policy**: questions about homework assignments sent less than 24 hours before submission deadline won't be answered.

- **Exception**: bug reports.

Peer Tutor

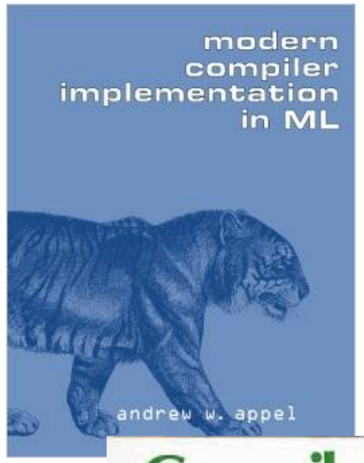
Tram Hoang

tram.hoang@u.yale-nus.edu.sg

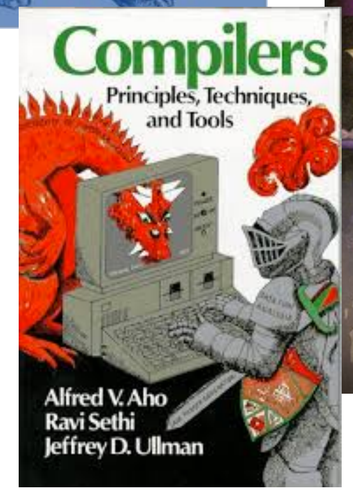
- Office Hours and Location TBA

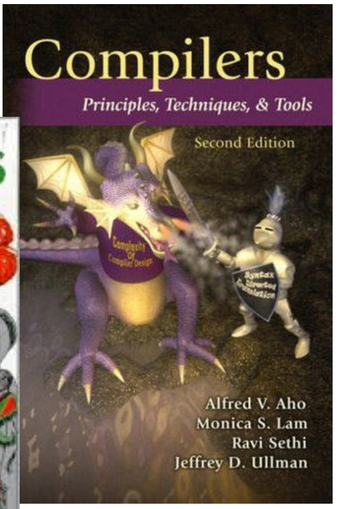
Resources

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

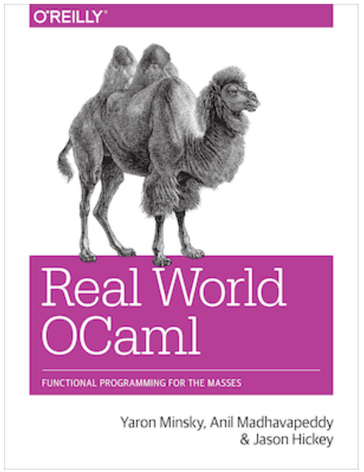


- Additional compilers books:
  - *Compilers – Principles, Techniques & Tools*
    (Aho, Lam, Sethi, Ullman)
    - *a.k.a. "The Dragon Book"*
  - *Advanced Compiler Design & Implementation* (Muchnick)





- About Ocaml:
  - *Real World Ocaml*
    (Minsky, Madhavapeddy, Hickey)
    - realworldocaml.org
  - *Introduction to Objective Caml* (Hickey)



OCaml, again!

- 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 YSC2229?
  - First two projects will help you get up to speed programming
  - See “Introduction to Objective Caml” by Jason Hickey
    - book available on the module web page, referred to in HW1

HW1: Hellocaml

- Homework 1 is available on Canvas
  - Individual project – no groups (the only in this module)
  - Due: *Wednesday, 25 August 2021 at 2:00am*
  - Topic: OCaml programming, an introduction to basic interpreters
  - Those who took YSC1212 with Prof. Danvy will find it *very* familiar

- Recommended software:
  - VSCode + OCaml extension
  - See the prerequisites page for the full setup

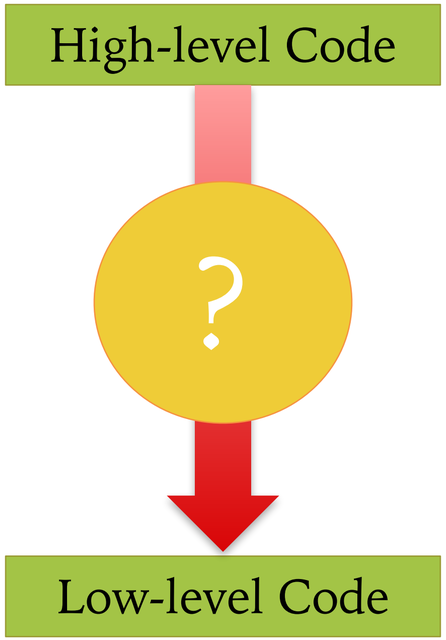
Any questions?

What is a Compiler, formally?

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.
  - Assembly is a textual representation of machine codes
- 1951—1952: Grace Hopper
  - developed the A-0 system (Arithmetic Language version 0) for the UNIVAC I
  - She later contributed significantly to the design of COBOL
- 1957: FORTRAN compiler built at IBM
  - Team led by John Backus
- 1960's: development of the first bootstrapping compiler for LISP

  See https://en.wikipedia.org/wiki/Tombstone_diagram
- 1970's: language/compiler design blossomed

1980s: ML / LCF
1984: Standard ML
1987: Caml
1991: Caml Light
1995: Caml Special Light
1996: Objective Caml
2005: F# (Microsoft)
2015: Reason ML

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

Source Code

- Optimized for human readability
  - *Expressive*: matches human ideas of grammar / syntax / meaning
  - *Redundant*: more information than needed (why?)
  - *Abstract*: exact computation on CPU possibly not fully determined by code
- Example C source:

```
#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

```
_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
```

- Optimized for Hardware
  - Mimics the logic of a particular processor (CPU): x86, Arm
  - Machine code hard for people to read
  - Redundancy, ambiguity reduced
  - Abstractions & information about intent is lost
- Assembly language
  - strong correspondence between the instructions in the language and the architecture's machine code instructions
  - text representation of the machine language
  - Etymology: internal instructions of a computer are “assembled” into the actual form used by the machine
- Figure at left shows (unoptimised) 32-bit code for the factorial function written in x86 assembly

How to translate?

- Source code – Machine code mismatch
- Some languages are farther from machine code than others:
  - Consider: C, C++, Java, Lisp, ML, Haskell, R, 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 *must* 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.
    (search for CompCert, Verified Software Toolchain, or Vellvm)

Specifying Compilers

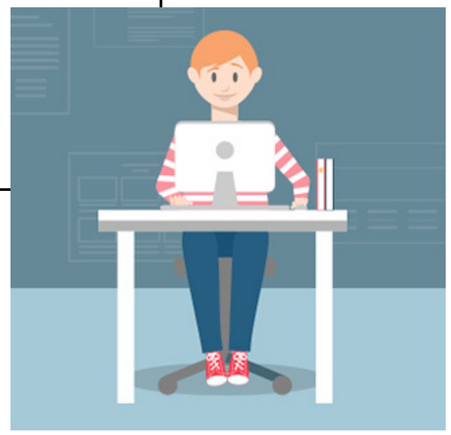
Program in C

```
#include <stdio.h>

#define IN  1    /* inside a word */
#define OUT 0    /* outside a word */

/* count lines, words, and characters in input */
main()
{
    int c, nl, nw, nc, state;

    state = OUT;
    nl = nw = nc = 0;
    while ((c = getchar()) != EOF) {
        ++nc;
        if (c == '\n')
            ++nl;
        if (c == ' ' || c == '\n' || c == '\t')
            state = OUT;
        else if (state == OUT) {
            state = IN;
            ++nw;
        }
    }
    printf("%d %d %d\n", nl, nw, nc);
}
```



compile

Program in x86 Assembly

```
792415C0    55          push ebp
792415C1    89E5        mov ebp, esp
792415C3    8B45 08     mov eax, [ebp+0x08]
792415C6    DB28        fld tword [eax]
792415C8    8B4D 0C     mov ecx, [ebp+0x0C]
792415CB    DB29        fld tword [ecx]
792415CD    DEC1        faddp
792415CF    8B55 10     mov edx, [ebp+0x10]
792415D2    DB3A        fstp tword [edx]
792415D4    DB68 0A     fld tword [eax+0x0A]
792415D7    DB69 0A     fld tword [ecx+0x0A]
792415DA    DEC1        faddp
792415DC    DB7A 0A     fstp tword [edx+0x0A]
                        pop ebp
                        ret 0x000C
```

Program P in C

```
#include <stdio.h>

#define IN  1    /* inside a word */
#define OUT 0    /* outside a word */

/* count lines, words, and characters in input */
main()
{
    int c, nl, nw, nc, state;

    state = OUT;
    nl = nw = nc = 0;
    while ((c = getchar()) != EOF) {
        ++nc;
        if (c == '\n')
            ++nl;
        if (c == ' ' || c == '\n' || c == '\t')
            state = OUT;
        else if (state == OUT) {
            state = IN;
            ++nw;
        }
    }
    printf("%d %d %d\n", nl, nw, nc);
}
```

compile

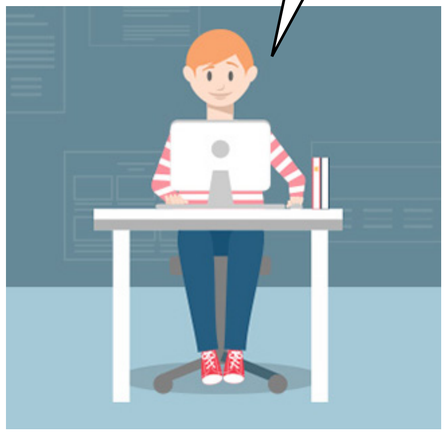
Program *compile*(P) in x86 Assembly

```
792415C0    55          push ebp
792415C1    89E5        mov ebp, esp
792415C3    8B45 08     mov eax, [ebp+0x08]
792415C6    DB28        fld tword [eax]
792415C8    8B4D 0C     mov ecx, [ebp+0x0C]
792415CB    DB29        fld tword [ecx]
792415CD    DEC1        faddp
792415CF    8B55 10     mov edx, [ebp+0x10]
792415D2    DB3A        fstp tword [edx]
792415D4    DB68 0A     fld tword [eax+0x0A]
792415D7    DB69 0A     fld tword [ecx+0x0A]
792415DA    DEC1        faddp
792415DC    DB7A 0A     fstp tword [edx+0x0A]
792415DF    5D          pop ebp
792415E0    C2 0C00     ret 0x000C
```

interpret-as-C

interpret-as-x86

Result(P, input) = $\mathbf{R_c}$ = $\mathbf{R_{x86}}$ = **Result**(*compile*(P), input)



Compiler **Specification:**

For *any* program P, and *any* input,
the result of *interpreting* P with input in **C** is the same as
the result of *executing compilation* of P with input in **x86 Assembly**.

or, equivalently

Correctness Theorem:

$$\forall \text{ P, input, } interpret_{C}(\text{P, input}) = execute_{x86}(compile(\text{P}), \text{input})$$

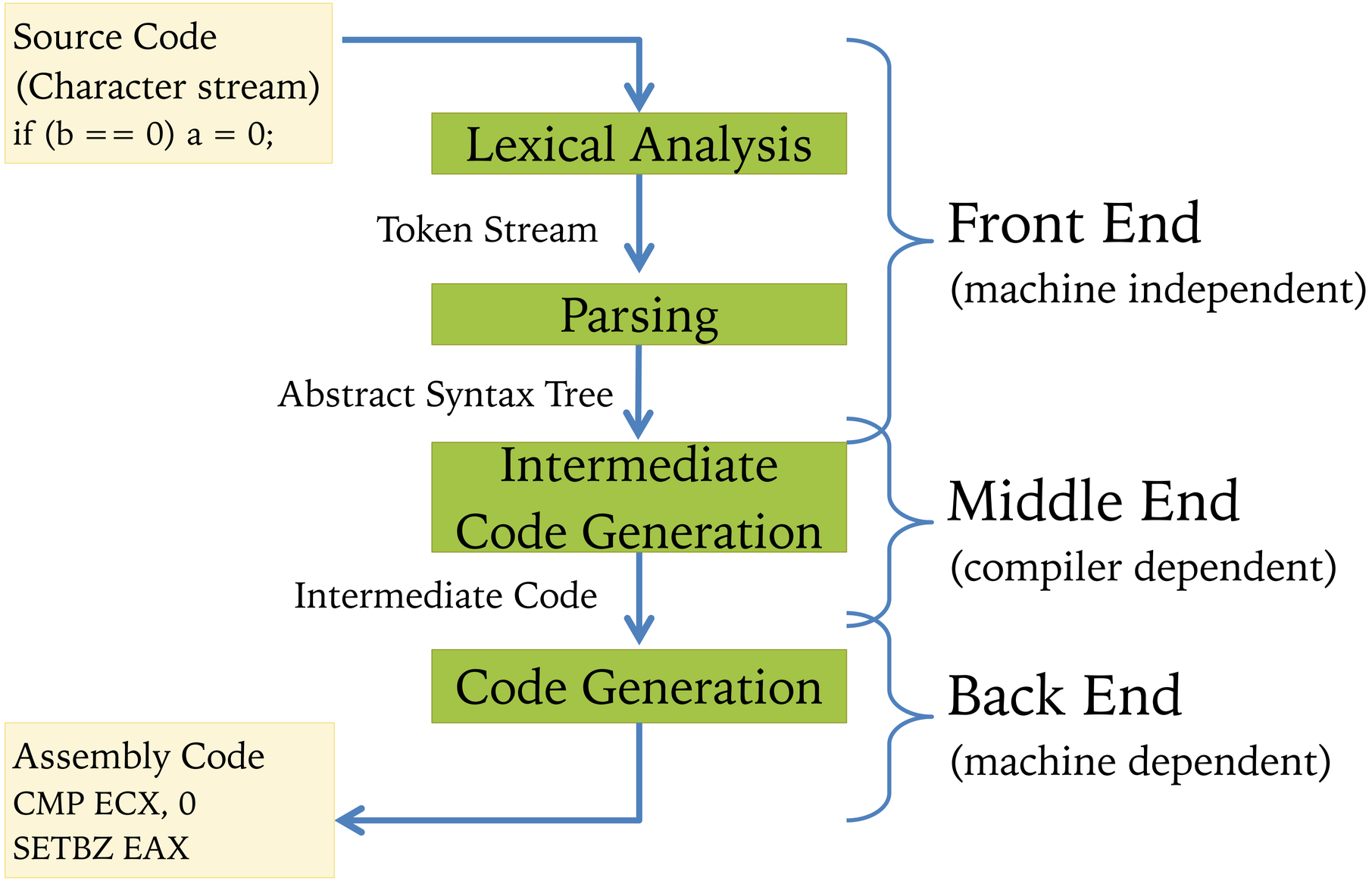
Idea: Translate in Steps

- Compile via a series of program representations

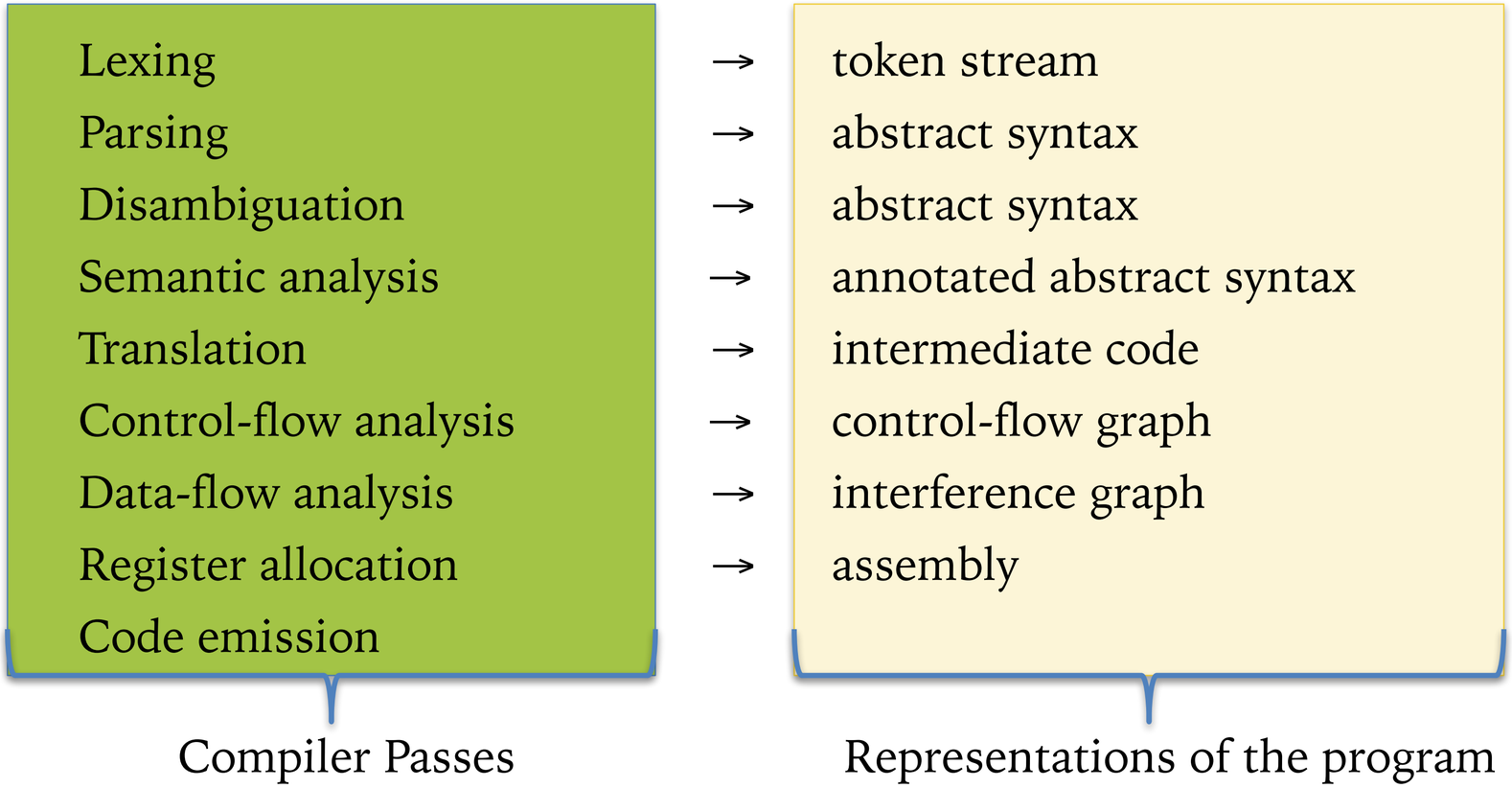
- Intermediate representations are optimised for program manipulation of various kinds:
  - Semantic analysis: type checking, error checking, etc.
  - Optimisation: dead-code elimination, common subexpression elimination, function inlining, register allocation, etc.
  - Code generation: instruction selection

- Representations are more machine specific, less language specific as translation proceeds

Simplified Compiler Pipeline

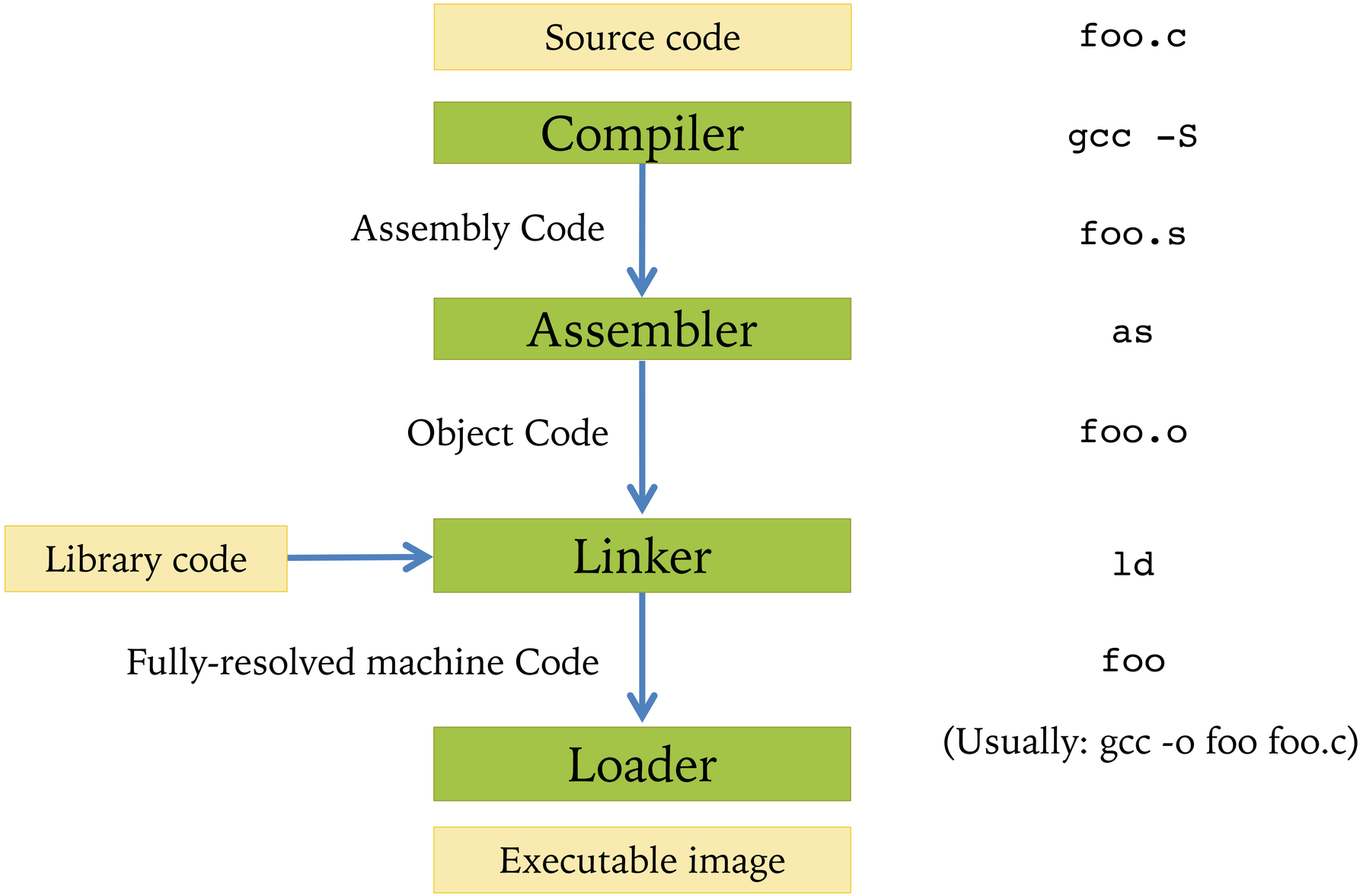


Typical Compiler Stages



- Optimisations 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 (processors may optimise, too)

Compilation and Execution



Compiler Demo

https://github.com/ysc4230/week-01-simple-2021

See `factorial.c` in the project root

(use `hexdump` to see binary files)

Short-term Plan

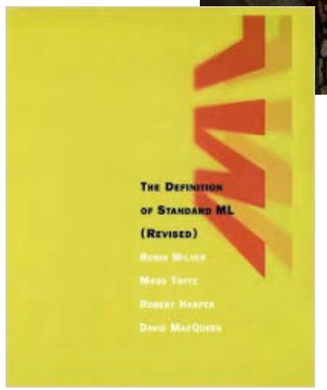
- Rest of today:
  - Refresher / background on OCaml
  - “object language” vs. “meta language”
  - Build a simple interpreter

- Next week:
  - Diving into x86 Assembly programming

OCaml for Compiler Hackers

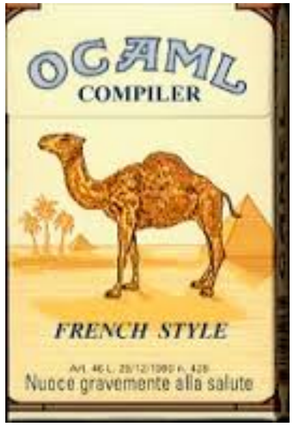
ML's History

- **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 Rechereche 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…
- **2005:** Microsoft's F# language is a descendent of OCaml
- **2013:** ocaml.org



OCaml Toolchain

- ocaml – the top-level interactive loop
- ocamlc – the bytecode compiler
- ocamlopt – the native code compiler
- ocamldep – the dependency analyser
- ocamldoc – the documentation generator
- ocamllex – the lexer generator
- ocamlyacc – the parser generator
- ocamlbuild – a compilation manager

- menhir – a more modern parser generator
- dune – a more compilation manager (build tool)
- 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 let
  - Has relatively few “side effects” (imperative updates to memory)

- Strongly & Statically typed
  - Compiler type-checks 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

Example: Imperative BST

```
type 'a node =
  | Node of (int * 'a ref * 'a tree * 'a tree)
  | Leaf
and 'a tree = ('a node) ref
let insert key value tree =
  let current = ref tree in
  let continue = ref true in
  while !continue do
    match !(!current) with
    | Leaf ->
      (!current) := Node (key, ref value, ref Leaf, ref Leaf)
    | Node (k, v, left, right) ->
      if k = key then begin
        v := value;
        continue := false;
      end else if k < key then
        current := left
      else
        current := right
  done
```

Example: Functional BST

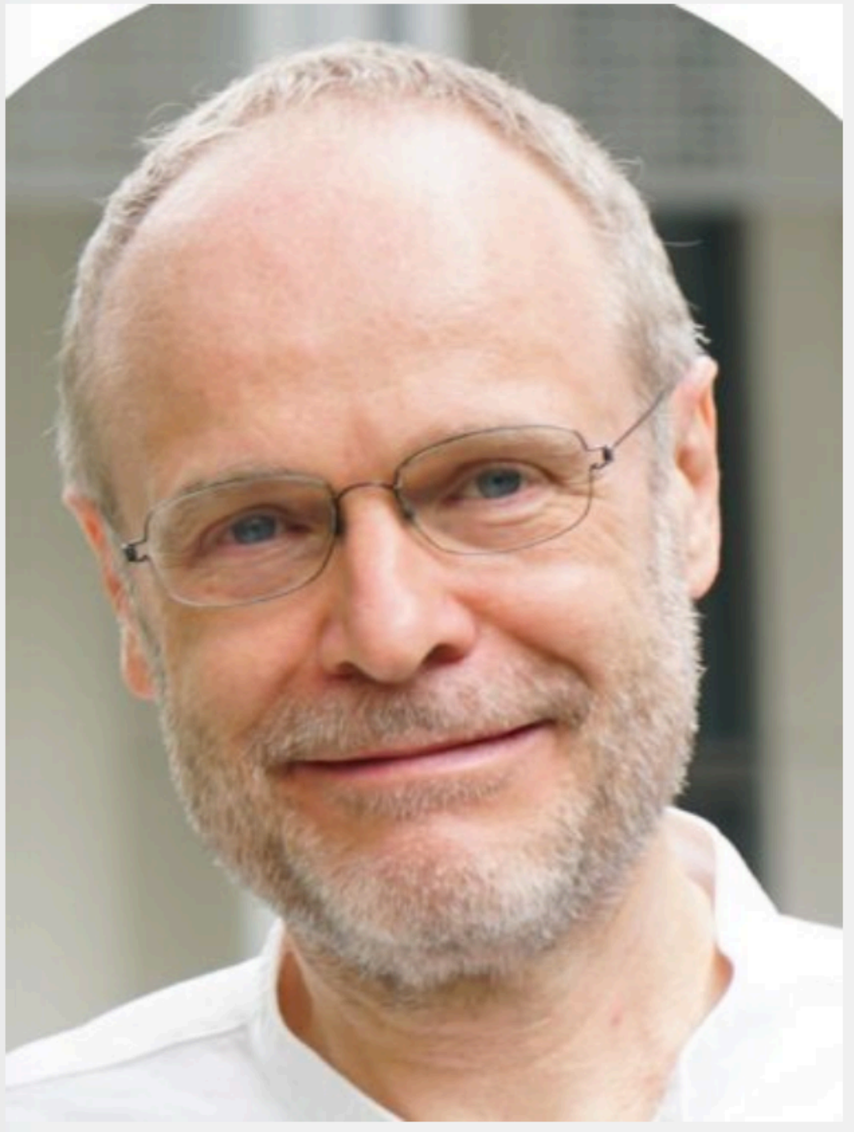
```
type 'a tree =
  | Node of (int * 'a * 'a tree * 'a tree)
  | Leaf
let rec insert key value tree =
  match tree with
  | Leaf -> Node (key, value, Leaf, Leaf)
  | Node (k, v, left, right) ->
    if k = key then
      Node (k, value, left right)
    else if k < key then
      Node (k, v, insert key value left, right)
    else
      Node (k, v, left, insert key value right)
```

Most Important OCaml Features for PLDI

- 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

Interpreters

- How to represent programs as data structures.
- How to write *programs* that *process programs*.



Olivier Danvy

Science (Computer Science)

Professor

Email: danvy@yale-nus.edu.sg

VIEW CURRICULUM VITAE

BIO | RESEARCH | PUBLICATIONS | TEACHING MODULES

Prof Danvy is interested in all aspects of programming languages, from their logic and semantics to their implementation, including programming, transforming programs, program transformations, and reasoning about programs and about program transformations (for one man's program is another program's data). As a Scheme programmer, he is familiar with parentheses and he is not afraid to use them. Also, for several years now, he has become convinced that the Coq Proof Assistant is the greatest thing since sliced bread and that it has the potential to transcend Computer Science college education, so watch this space. He is also interested in scientific communication.

Everyone's Favorite Function

- Consider this implementation of factorial in a hypothetical programming language that we'll call "SIMPLE"
  (Simple IMperative Programming LanguagE):

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

- We need to describe the constructs of this SIMPLE
  - ***Syntax:*** which sequences of characters count as a legal "program"?
  - ***Semantics:*** what is the meaning (behavior) of a legal "program"?

"Object" vs. "Meta" language

Object language:
the language (syntax / semantics) being described or manipulated

Metalanguage:
the language (syntax / semantics) used to *describe* some object language

Today's example:

SIMPLE	interpreter written in OCaml

Course project:

OAT ⇒ LLVM ⇒ x86asm	compiler written in OCaml

Clang compiler:

C/C++ ⇒ LLVM ⇒ x86asm	compiler written in C++

Metacircular interpreter:

lisp	interpreter written in lisp

Grammar for a Simple Language

```
<exp> ::=
    |   <X>
    |   <exp> + <exp>
    |   <exp> * <exp>
    |   <exp> < <exp>
    |   <integer constant>
    |   (<exp>)

<cmd> ::=
    |   skip
    |   <X> = <exp>
    |   ifNZ <exp> { <cmd> } else { <cmd> }
    |   whileNZ <exp> { <cmd> }
    |   <cmd>; <cmd>
```

BNF grammars are themselves domain-specific metalanguages for describing the syntax of other languages…

- Concrete syntax (grammar) for the Simple language:
  - Written in "Backus-Naur form"
  - <*exp*> and <*cmd*> are ***nonterminals***
  - '::=' , '|' , and <…> symbols are part of the m*etalanguage*
  - 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)

Demo: Interpreters in OCaml

- https://github.com/ysc4230/week-01-simple
- Interpreting expressions
- Translating Simple programs to OCaml programs

Next Week

- Basics X86 Assembly
- C memory layout
- Implementing calls and returns via call stacks