Lecture 1

CIS 341: COMPILERS

Administrivia

- Instructor: Steve Zdancewic
  Office hours: Thurs. 4:00-5:00 & by appointment
  Levine 511

- TAs:
  - Dmitri Garbuzov
  - Hongbo (Bob) Zhang
    Office hours: To be determined

- E-mail: cis341@seas.upenn.edu
- Web site: http://www.seas.upenn.edu/~cis341
- Piazza: http://piazza.com/upenn/spring2013/cis341

Resources

- Course textbook: (recommended, not required)
  - *Modern compiler implementation in ML* (Andrew Appel)
- Additional books:
  - *Compilers – Principles, Techniques & Tools* (Aho, Lam, Sethi, Ullman)
    - *a.k.a. "The Dragon Book"*
  - *Advanced Compiler Design & Implementation* (Muchnick)

- Acknowledgments:
  - Some CIS 341 material modeled on courses by Andrew Myers (at Cornell) and Greg Morrisett (at Harvard)

Course Policies

- Prerequisites: CIS121 and CIS240
  - Significant programming experience

Grading:

- 72% Projects: *The Quaker OAT Compiler*
  - Groups of 1 or 2 students
  - Implemented in OCaml

- 12% Midterm
- 16% Final exam

- Lecture attendance is crucial
- *No laptops!* (It's too distracting for me and for others in the class.)

Project Policies

- Projects (except Project 0) may be done individually or in pairs
- Late projects:
  - Projects may be turned in up to 2 days late at a cost of 10% penalty per day to the project grade.
  - If you have a legitimate excuse, extensions may be given with prior approval of the course instructor.
- 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!*

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
  - 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 up to *tens* of hours per week…

The Quaker OAT Compiler

- Course projects
  - Project 0: OCaml Programming
  - Project 1: X86lite interpreter
  - Project 2: Parsing & Basic code generation (the "A" language)
  - Project 3: Control flow (the "T" language)
  - Project 4: Procedures & arrays (the "AT" language)
  - Project 5: Objects & type checking (the "OAT" language)
  - Project 6: Optimizations
  - Project 7: OAT programming

- Goal: build a complete compiler from an object-oriented type-safe language to x86 assembly.

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

- No experience with OCaml is necessary
  - Next couple lectures 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 pages, referred to in Project0

Project 0: Hellocaml

- Project 0 available later today on the course web site.
  - Individual project – no groups
  - Due: Tuesday, 22 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 "ocamlc" to run the compiler

- We recommend using either:
  - Eclipse with the OcaIDE plugin
  - Emacs
  - See the course web pages to get started

Announcements

- Prof. Zdancewic will be out of town next Tuesday (Jan. 15th)
  - Class is *CANCELLED*
  - Use the spare time to brush up on OCaml

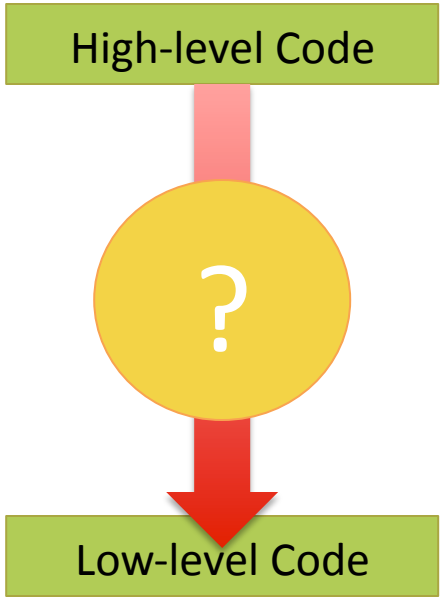
- TAs may have "getting started" office hours during class
  - Watch Piazza for news

What is a compiler?

COMPILERS

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, Java bytecode JIT compilers, etc.



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...

Source Code

- 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 OAT source:

```
class Point <: Object {
  int x;
  int y;
  new ()() {
    this.x = 0;
    this.y = 0
  }
  Point move(int dx, int dy) {
    this.x = this.x + dx;
    this.y = this.y + dy;
    return this;
  }
}
(new Point()).move(3,4).x
```

Low-level code

- Optimized for Hardware
  - Machine code hard for people to read
  - Redundancy, ambiguity reduced
  - Abstractions & information about intent is lost
- Machine code ≈ Assembly
- Figure at right shows (unoptimized) code for the Point.move method

```
.globl __fun__Point.move
__fun__Point.move:
        pushl %ebp
        movl %esp, %ebp
        subl $4, %esp
__5:
        movl 8(%ebp), %eax
        movl 4(%eax), %eax
        movl %eax, -4(%ebp)
        movl 12(%ebp), %ecx
        addl %ecx, -4(%ebp)
        movl -4(%ebp), %ecx
        movl 8(%ebp), %eax
        movl %ecx, 4(%eax)
        movl 8(%ebp), %eax
        movl 0(%eax), %eax
        movl %eax, -4(%ebp)
        movl 16(%ebp), %ecx
        addl %ecx, -4(%ebp)
        movl -4(%ebp), %ecx
        movl 8(%ebp), %eax
        movl %ecx, 0(%eax)
        movl 8(%ebp), %eax
        movl %ebp, %esp
        popl %ebp
        ret
```

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
  - There is much ongoing research about *proving* compilers correct. (Google for CompCert, Verified Software Toolchain, or Vellvm)

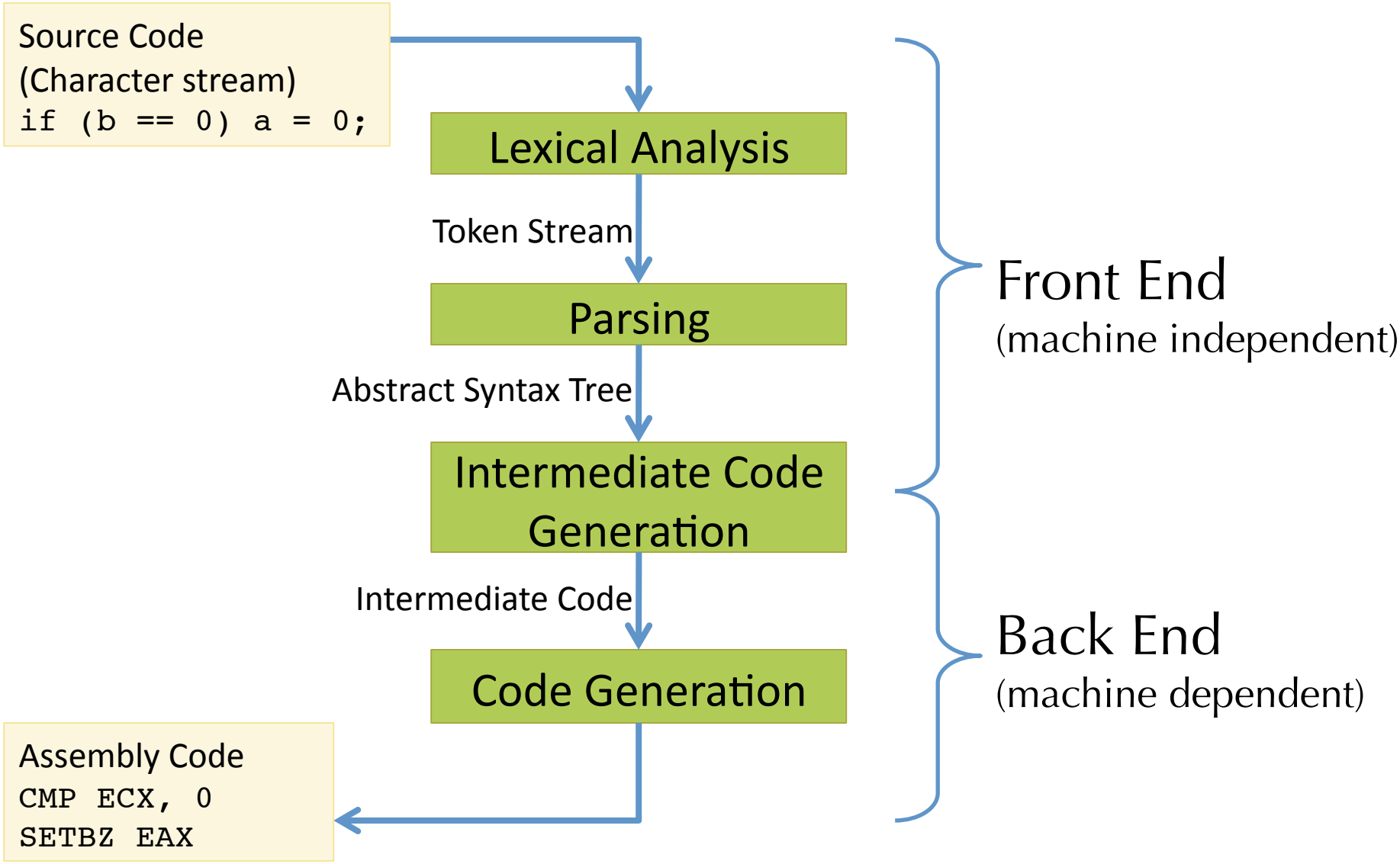
Idea: Translate in Steps

- 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

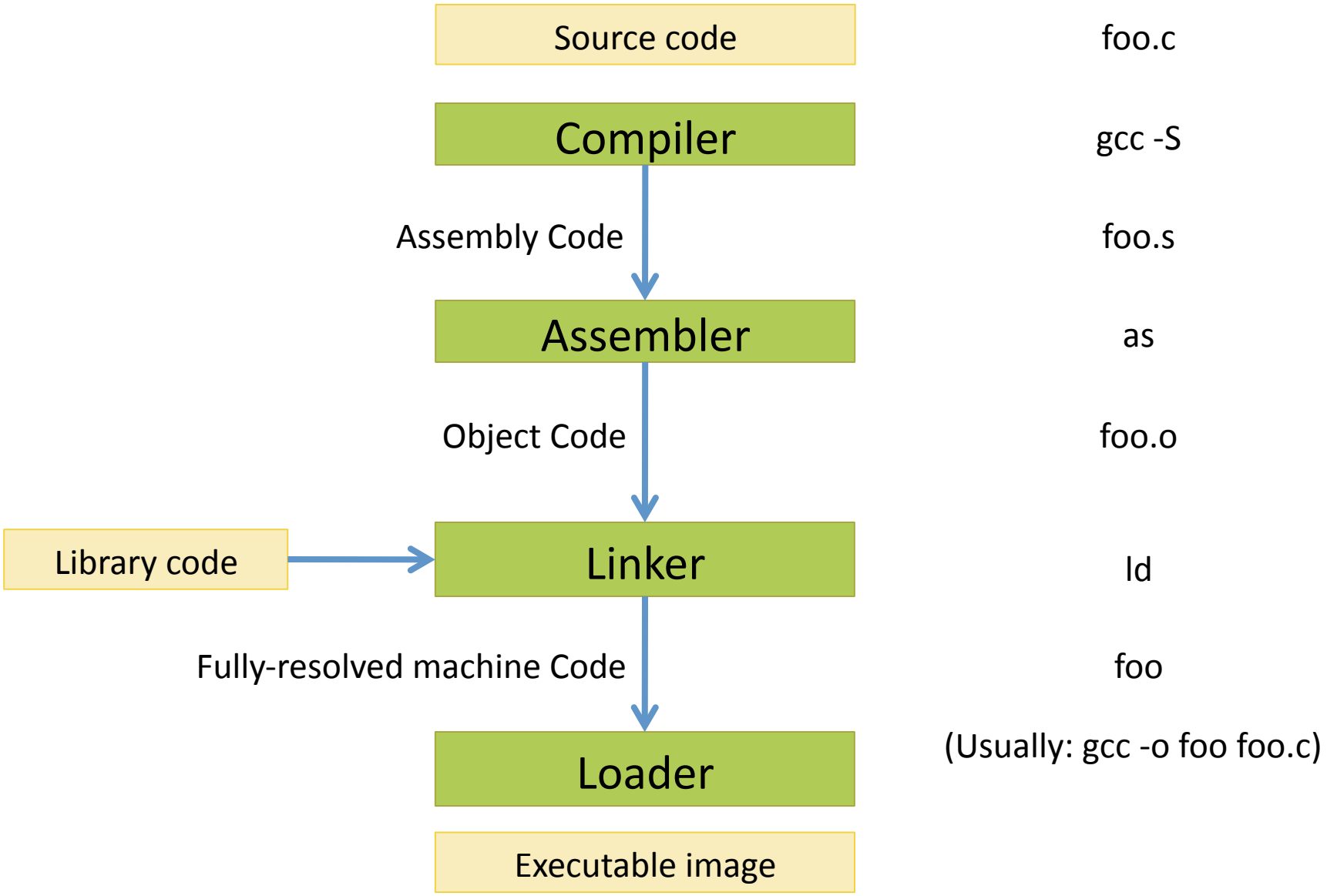


Typical Compiler Stages

- 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



Introduction to OCaml programming

- A little background about ML
- Interactive tour via the OCaml top-loop & Eclipse
- Writing simple interpreters

OCAML

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
- ML escaped into the wild and became "Standard ML" 1984
  - 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 – being discussed intermittently

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!)
- 1994: Development of CAML passed to project Cristal
- 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

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
- ocamlbuild – a compilation 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 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*

Most Important Features for CIS341

- Types:
  - int, bool, int32, char, string, built-in lists, tuples, records, functions
- Concepts:
  - Pattern matching
  - Recursive functions over algebraic datatypes
- Libraries:
  - Int32, List, Printf, Format

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

INTERPRETERS

Factorial: Everyone's Favorite Function

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

```
ans = 1;
whileNZ (x) {
    ans = ans * x;
    x = x-1
}
```

- We need:
  - A way to describe the constructs of this hypothetical language
  - A way to describe the language semantics

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>
```

- Concrete syntax (grammar) for a simple imperative language
  - Written in "Backus-Naur form"
  - <*exp*> and <*cmd*> are *nonterminals*
  - '::=' and '|' 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