There are different ways of translating a programming language

- **Source Code**
  - Compiler
  - Binary/Assembly
  - Ex.: C, C++

- **Source Code**
  - Interpreter
  - Ex.: Python

- **Source Code**
  - Compiler
  - Bytecode
  - VM
  - Ex.: Java
“Go straight on to the roundabout; mind the lorries”

“OK, so keep going that way”; “Here, you’re going to go straight ahead”

“It means ‘keep going until you get to this circular intersection; watch out for trucks.’”
Compilers vs. interpreters

• Compiler
  • Translates the program to a form executable by the machine (or assembly)
  • Compile, then can run the executable: compiler no longer involved

• Interpreter
  • Doesn’t translate to machine-readable format
    • Might compile to bytecode or intermediate representation
  • Runs ("interprets") program directly
  • Can’t run without the interpreter
Compilers translate code in phases

Source Code → Lexical Analyzer → Tokens → Parser → Abstract Syntax → Lowering → Intermediate Representation → Code Generator → Target Code

```
a = b + c - 1
```

```
VAR a
EQUAL
VAR b
OP +
VAR c
OP -
CONST 1
```

```
Assign
  a
+
  b
-
  c
1
```

```
temp = c - 1
a = b + temp
```

```
subl %rax, 1
addl %rax, %rbx
```
May have many more phases, several intermediate representations

- WordLang: imperative language with machine words, memory and
- DataLang: imperative language
- BVI: one global variable
- BVL: functional language without closures
- GOOSLang: last language with closures (has multi-arg closures)

- Force two-reg code (if req.)
- Perform SSA-like renaming
- Select target instructions
- Simplify program
- Remove data abstraction
- Combine adjacent memory allocations
- Reduce caller-saved vars
- Switch to imperative style
- Optimise Let-expressions
- Compile global vars into a dynamically resized array
- Split over-sized functions into many small functions
- Fold constants and shrink Lets
- Remove deadcode
- Perform closure conv.
- Inline small functions
- Introduce C-style fast calls wherever possible
- Hack where closure values flow & inline small functions
Front End is language specific
Back End is machine specific

Source Code
Lexical Analyzer

Tokens
Parser

Abstract Syntax
Lowering

Analysis

Code Gen.
Target Code

“Front End”
“Middle End”
“Back End”

VAR a
EQUAL
VAR b
OP +
VAR c
OP -
CONST 1

Assign

temp = c - 1
a = b + temp

subl %rax, 1
addl %rax, %rbx

a = b + c - 1
Can (and usually do) swap out back ends to target different machines
Compiler collections also swap out front ends for different languages.

Intermediate Representation

Machine-Independent Optimizations

C
C++
Java

x86
ARM
PowerPC

...
• More about compilers: CS443
• This class: more about interpreters
Functional Programming

• Strong mathematical foundations
• Very high-level
• Really elegant for expressing many algorithms

(Alt text: Functional programming combines the flexibility and power of abstract mathematics with the intuitive clarity of abstract mathematics)
OCaml

• Statically typed, functional
  • (also has imperative and object-oriented features)

• Strong, expressive type system
  • (makes implementing many data structures very easy)

• Type inference
  • int x = 5;
  • x = 5
• Probably the most used functional language
• First appeared 1996
  • “ML family” of languages (Standard ML, F#) goes back to the 1970s
• Version 5.0.0 released Dec. 16, 2022

• Industrial-strength compiler
  • Actively maintained
  • Lots of libraries (standard and 3rd-party)