How Programming Languages Work (Basics)

Kenjiro Taura

2024/05/19

Contents

Introduction

Why you want to make a language, today?

- new hardware
  - GPUs, AI chips, Quantum, …
  - new instructions (e.g., SIMD, matrix, …)
- new general purpose languages
  - Scala, Julia, Go, Rust, etc.

Why you want to make a language, today?

- special purpose (domain specific) languages
  - statistics (R, MatLab, etc.)
  - data processing (SQL, NoSQL, SPARQL, etc.)
  - deep learning
  - constraint solving, proof assistance (Coq, Isabelle, etc.)
  - macro (Visual Basic (MS Office), Emacs Lisp (Emacs), Javascript (web browser), etc.)

Taxonomy : interaction mode

- **interactive / read-eval-print-loop (REPL)**
  - type code directly or load source code in a file interactively
  - Julia
- **batch compile**
  - convert source into an executable file
  - and run it (typically the “main” function)
  - Go, Rust
- some language implementations provide both
  - OCaml

Taxonomy : execution strategy

- **interpreter** executes source code directly with its input
  - interpreter (*source-code*, *input*) $\longrightarrow$ *output*
- **compiler** first converts source code into ***a machine (assembly) code*** that is directly executed by the processor
  - compiler (*source-code*) $\longrightarrow$ *machine-code*;
  - *machine-code* (*input*) $\longrightarrow$ *output*
- **translator** or **transpiler** are like compiler, but convert into another language, not machine (assembly) code

Taxonomy : compiler/translator

- **ahead-of-time (AOT)** compiler converts all the program parts into assembly before execution
- **just-in-time (JIT)** compiler converts program parts incrementally as they get executed (e.g., a function at a time)

CPU and machine code : An overview

What a machine (assembly) code looks like

- it *is* just another programming language
- it has many features present in programming languages

high-level languages	**machine code**
variables	*registers* and *memory*
structs and arrays	memory and load/store instructions
expressions	arithmetic instructions
if / loop	compare / conditional jump instructions
functions	jump and link instructions

What a machine (assembly) code looks like

- compilation is nothing like a magic; it's more like translating English to French

What a CPU (core) looks like

- a small number (typically < 100) of **registers**
  - each register can hold a small amount of (e.g., 64-bit) data
- majority of data are stored in **memory** (a few to 1000 GB)

[Insert image here: cpu.pdf]

Memory

- where majority of data your program processes are stored
- memory is essentially a large flat array indexed by integers, often called **addresses**
- an address is just an integer

[Insert image here: cpu.pdf]

What a CPU (core) does

- a special register, called **program counter** or **instruction pointer**, specifies the address to fetch the next instruction at
- a CPU core is essentially a machine that does the following:

```
repeat:
  inst = memory[program counter]
  execute inst
```

- an instruction:
  - performs some computation on values in registers or memory

What a CPU (core) does

- changes the program counter (typically to next instruction)

[Insert image here: cpu.pdf]

Exercise objectives

- `pl06_how_it_gets_compiled`
- learn how a **compiler** does the job
- by inspecting assembly code generated from source language functions

A glance at x86 machine (assembly) code

The first glance

```
.file   "add123.go"
.section    .go_export,"",@progbits
...
.text
.globl  go_0pl06.Add123
.type   go_0pl06.Add123, @function
go_0pl06.Add123:
.LFB0:
  .cfi_startproc
  cmpq    %fs:112, %rsp
  jb      .L3
.L2:
```

The first glance

```
  leaq    123(%rdi), %rax
  ret
.L3:
  movl    $0, %r10d
  movl    $0, %r11d
  call    __morestack
  ret
  jmp     .L2
  .cfi_endproc
.LFE0:
  .size   go_0pl06.Add123, .-go_0pl06.Add123
```

The first glance

```
    .globl  go.pl06..types
  ...
```

> looks scary?

Unimportant lines

- indented lines starting with a dot (e.g., `.file`, `.section`, `.text`, etc.) are **not instructions** and largely not important
- lines ending in a colon (e.g., `.L2:`, `go_0pl06.Add123:`) are **labels** used as jump/call targets

Where to look

- focus on lines that are **instructions**
- instructions for a function start with a label **similar to** the function name (but not exactly due to name mangling)

Registers

- general-purpose 64-bit integer registers: `rax`, `rbx`, `rcx`, `rdx`, `rdi`, `rsi`, `r8`–`r15`, `rbp`
- floating-point registers: `xmm0`–`xmm15`
- stack pointer: `rsp`
- compare flag register: `eflags` (set/used implicitly)
- instruction pointer: `rip` (set by every instruction)

[ref: https://wiki.cdot.senecapolytechnic.ca/wiki/X86_64_Register_and_Instruction_Quick_Start]

Frequently used instructions

- `addq`, `leaq`, `subq`, `imulq`, `idivq`
- `movq`: move between registers/memory (load/store)
- `cmpq`: compare and set flags in `eflags`
- conditional jumps: `jl`, `jle`, `jg`, `jge`, `je`, `jne`
- `call`, `ret`: call or return from function

Reading instructions and operands

e.g.,

```
addq x, y  →  y += x
opq  x, y  ≡  y = y op x
subq x, y  ≡  y = y - x
```

Syntax of operands

- `$n` = immediate value
- `%R` = register named `R`
- `(...)` = address operand

Examples:

- `addq $1, %rax` → add 1 to rax
- `subq $1, %rax` → subtract 1 from rax

Address operands

Forms:

- `(%R)` → value at address in R
- `n(%R)` → value at address `R + n`
- `n(%R, s, %R')` → address `R + s * R' + n`

Examples:

- `mulq (%rdi), %rax` → multiply `rax` by value at address in `rdi`
- `movq %rax, 8(%rdi)` → store rax to rdi + 8
- `leaq 16(%rdi,8,%rsi), %rax` → `rax = rdi + 8 * rsi + 16`

Julia assembly syntax

- syntax and operand order differ between assemblers
- output from Julia (`code_native`) uses **destination-first syntax**

```
addq x, y  ≡  x += y
```

| GNU | Julia | |

Julia assembly syntax

|

Julia assembly syntax

-||`mulq (%rdi), %rax`|`mulq %rax, [%rdi]`||`movq %rax, 8(%rdi)`|`movq [%rdi+8], %rax`||`leaq 16(%rdi,8,%rsi),%rax`|`leaq %rax, [%rdi+8*%rsi+16]`|

Things to learn in the exercise

1. Calling convention / ABI : How parameters and return values are passed (typically via registers)
2. Data representation : Learn how data types (ints, floats, structs, pointers, arrays) are represented

   ```
   f(a, i) = a[i]
   ```

3. Control flow : How conditionals and loops are implemented
4. Function calls : How function call/return is implemented