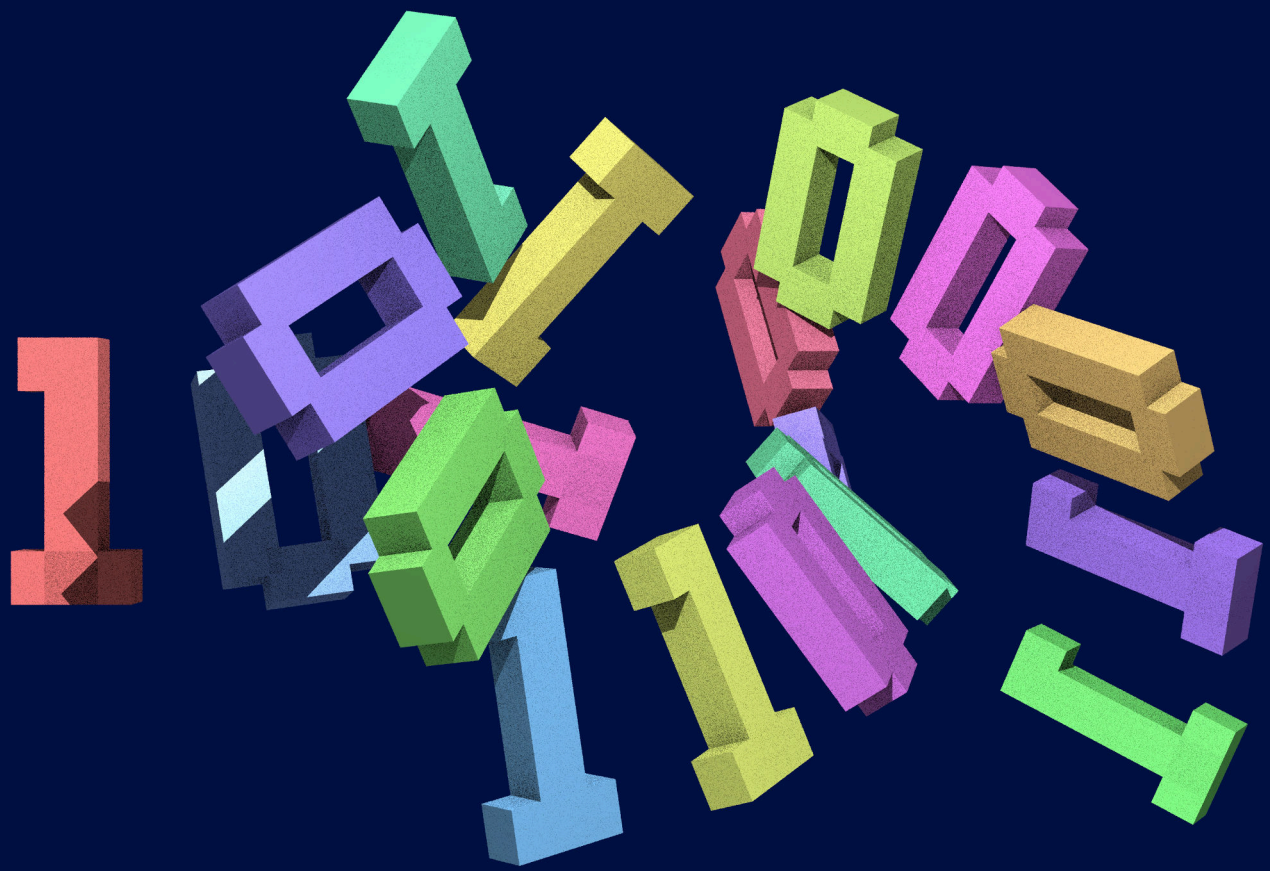
From Source Code To Machine Code

Build Your Own Compiler From Scratch

Compiler, Interpreter & Assembly



https://build-your-own.org

From Source Code To Machine Code

Build Your Own Compiler From Scratch

James Smith

build-your-own.org

2023-05-18

Contents

01. Introduction

1.1 Why Compiler?

Compilers are a fascinating topic. You use them every day. They turn your textual source code into some binary magic executed by computers. Ever wondered how they work? You can learn all of this by coding a mini-compiler by yourself, thus removing a mystery from your mind.

Compiler construction is a subject included in computer science and software engineering education. However, many of today's (2023+) coders do not have a formal CS education. Basic things such as compilers, databases, operating systems, etc. are often seen as magical black boxes. That's why I started the "*Build Your Own X*" book series. To learn and teach basic things by the "*from scratch*" approach, through succinct & condensed books.

It's also considered a fun side project that just cost you some free time.

1.2 Components of Compilers

Let's start with the source: some text files written in a programming language. To do anything useful with a source file (whether it's a compiler or an IDE), the text must be "parsed" into some structures. The jargon for this process is "parser".

The parser of a big language is often divided into 2 phases: first, the text is split into a stream of "tokens" (keywords, names, operators, think of the words in an article), then the parser assembles the tokens into structures (grammar). The first phase is often called a "tokenizer" or "scanner", although this phase is not necessary.

The parsed structure is often called a "syntax tree". One can simply simulate the program from the syntax tree. This is called an "interpreter", and it's a naive interpreter.

A less naive interpreter includes extra steps. Such as converting the syntax tree into a list of instructions called "bytecode". The execution of a program works on the bytecode instead of the syntax tree. Why the extra step? You'll find out later.

The component for producing the bytecode is also called a "compiler", and the line between compiler and interpreter blurs from here. A compiler such as GCC ultimately outputs a binary executable — a container for machine code — which also contains a list of instructions, the real instructions that the CPU executes.

There are often multiple steps between the syntax tree and the final machine code and some "intermediate representations" (IR) for various purposes. Bytecode can also be an IR. However, what we describe in this section are components of typical compilers and interpreters; there are also many alternative compiler/interpreter designs and implementations, which you can have fun exploring them.

1.3 The Book

Like my other books, this book follows a step-by-step approach. You start with simple things, like a calculator, then a simple interpreter. Then you tackle the compiler bit-by-bit, compiling to bytecode, learning assembly, and finally generating native executables.

The final product is a small, statically typed language that resembles a subset of C, which compiles into x64 Linux ELF executables. And it has a machine-agnostic IR, so it's not hard to target other platforms such as ARM, which is a challenge you can take up later.

The project is written from scratch in pure Python without any dependencies, although everything is language agnostic. You can find my other books on the official website: https://build-your-own.org

02. A Simple Calculator

2.1 Introduction

The first step in building a compiler or interpreter is to parse the source code into tree structures. Most parsing can be done recursively, and you can parse most computer languages once you learn how to do it.

Parsing is the boring part of a compiler, and most production programming languages require a lot of labor work to parse. Fortunately, there is a type of grammar that can be parsed with minimal effort: the S-expression (sexpr). We'll use it to save the time.

2.2 The Sexpr

An sexpr is either a nested list enclosed in parentheses, or an indivisible element called an "atom". Some examples:

```
;; atom
a
123
;; list
(a)
(a b c d)
;; nested list
(a ((b) c))
```

It naturally represents a tree, and most parsing is converting a string representation into a tree structure, so why not just use the sexpr for syntax? Especially when the sexpr is simpler and easier to parse than alternatives.

Besides the obvious tree structure, sexpr-based syntax often uses prefix operators instead of infix operators. Consider the following cases:

Infix operators:

```
1 + 2 * 3
```

Prefix operators:

```
(+ 1 (* 2 3))
```

Infix operators are complicated by different operator precedence — a problem that doesn't exist with prefix operators. Prefix operators are also easier to handle when we need further parsing. That's why I use sexpr in this book. If you like challenges, you can try using an alternative grammar, such as a C-like grammar with infix operators.

We'll implement the following operators:

1. Binary operators: `(op a b)`.

   Arithmetics:

   - `(+ 1 2)`
   - `(- 1 2)`
   - `(* 1 2)`
   - `(/ 1 2)`

   Comparisons:

   - `(lt 1 2)`, less than.
   - `(gt 1 2)`, greater than.
   - `(eq 1 2)`, equal.
   - `(ne 1 2)`, not equal.

   Booleans:

   - `(and a b)`
   - `(or a b)`

2. Unary operators:

   - `(- a)`, negative.
   - `(not a)`, boolean not.

3. Conditional: `(? expr-cond expr-then expr-else)`.

   e.g., `(? (lt 1 2) "yes" "no")` yields "yes".

4. `(print a b c)`

2.3 Start Coding

An sexpr will be parsed into either a Python list or a Python string. For example, the sexpr `(a b (+ 1 2))` will be parsed into:

```
["a", "b", ["+", ["val", 1], ["val", 2]]]
```

Note that numbers or strings in the sexpr are wrapped by the `["val", x]` structure unlike the symbols `a` or `b`, this is explained later.

The function `parse_expr` take the input string and the current offset (`idx`) into the input as arguments. It advances the offset during parsing until an error is encountered or the sexpr is terminated.

To parse an sexpr, the first step is to determine whether it's an atom or a list, this is done by looking at the first non-whitespace character.

```
def parse_expr(s: str, idx: int):
    idx = skip_space(s, idx)
    if s[idx] == '(':
        # a list
        ...
    elif s[idx] == ')':
        raise Exception('bad parenthesis')
    else:
        # an atom
        ...
```

If it's a list, parse recursively until the closing parenthesis is met.

```
def parse_expr(s: str, idx: int):
    idx = skip_space(s, idx)
    if s[idx] == '(':
        # a list
        idx += 1
        l = []
        while True:
            idx = skip_space(s, idx)
            if idx >= len(s):
```

```
                raise Exception('unbalanced parenthesis')
            if s[idx] == ')':
                idx += 1
                break

            idx, v = parse_expr(s, idx)
            l.append(v)
        return idx, l
    elif s[idx] == ')':
        raise Exception('bad parenthesis')
    else:
        # an atom
        start = idx
        while idx < len(s) and (not s[idx].isspace()) and s[idx] not in '()':
            idx += 1
        if start == idx:
            raise Exception('empty program')
        return idx, parse_atom(s[start:idx])
```

Any non-whitespace character is considered part of an atom.

```
def skip_space(s, idx):
    while idx < len(s) and s[idx].isspace():
        idx += 1
    return idx
```

Since we are going to evaluate the parsed sexpr, it makes sense to extract values such as numbers and strings from atoms. The "val" string at the list head is for distinguishing parsed values from other atoms (symbol).

```
# bool, number, string or a symbol
def parse_atom(s):
    # TODO: actually implement this
    import json
    try:
        return ['val', json.loads(s)]
    except json.JSONDecodeError:
        return s
```

The above function is just a placeholder, we'll ditch it in later chapters.

As we can see, the pattern of parsing is:

1. Look at what's at the beginning and decide on the next step.
2. Recurse if necessary.

Even for more complicated languages such as C or Java, their grammar can be parsed in this way. Although this might not be obvious to you at this point.

And finally, we need to check that the input is fully exhausted:

```
def pl_parse(s):
    idx, node = parse_expr(s, 0)
    idx = skip_space(s, idx)
    if idx < len(s):
        raise ValueError('trailing garbage')
    return node
```

2.4 Expression Evaluation

Evaluating an sexpr is dead simple:

1. Look at the first item in the list and decide what the operation is.
2. Evaluate the rest of the items recursively.
3. Perform the operation.

```
def pl_eval(node):
    if len(node) == 0:
        raise ValueError('empty list')

    # bool, number, string and etc
    if len(node) == 2 and node[0] == 'val':
        return node[1]

    # binary operators
    import operator
    binops = {
```

```
        '+': operator.add,
        '-': operator.sub,
        '*': operator.mul,
        '/': operator.truediv,
        'eq': operator.eq,
        'ne': operator.ne,
        'ge': operator.ge,
        'gt': operator.gt,
        'le': operator.le,
        'lt': operator.lt,
        'and': operator.and_,
        'or': operator.or_,
    }
    if len(node) == 3 and node[0] in binops:
        op = binops[node[0]]
        return op(pl_eval(node[1]), pl_eval(node[2]))

    # unary operators
    unops = {
        '-': operator.neg,
        'not': operator.not_,
    }
    if len(node) == 2 and node[0] in unops:
        op = unops[node[0]]
        return op(pl_eval(node[1]))

    # conditionals
    if len(node) == 4 and node[0] == '?':
        _, cond, yes, no = node
        if pl_eval(cond):
            return pl_eval(yes)
        else:
            return pl_eval(no)

    # print
    if node[0] == 'print':
        return print(*(pl_eval(val) for val in node[1:]))

    raise ValueError('unknown expression')
```

It's basically mapping operators to Python operators. The expression evaluator can be used as an (unconventional) calculator.

```
def test_eval():
    def f(s):
        return pl_eval(pl_parse(s))
    assert f('1') == 1
    assert f('(+ 1 3)') == 4
    assert f('(? (lt 1 3) "yes" "no")') == "yes"
    assert f('(print 1 2 3)') is None
```

We have implemented a simple calculator, which is the basis of the next chapter project — a programming language with variables, control flows, and functions.

03. Interpreter: Variables and Scopes

3.1 Introduction

In the last chapter we implemented a simple calculator. To make it look like a programming language we need to add 3 more aspects:

1. Variables. For manipulating states.
2. Control flows. Like conditionals and loops.
3. Functions. For reusing code.

Here are some sample programs that demonstrate the syntax of our language:

```
;; define the function `fib` with an argument `n`
(def fib (n)
    ;; if-then-else
    (if (le n 0)
        (then 0)
        (else (+ n (call fib (- n 1))))))   ;; function call

(call fib 5)
```

```
;; another version of the `fib` function
(def fib (n) (do
    (var r 0)           ;; variable declaration
    (loop (gt n 0) (do  ;; loop
        (set r (+ r n)) ;; variable assignment
        (set n (- n 1))
    ))
    (return r)          ;; return from a function call
))

(call fib 5)
```

3.2 New Commands for Variables

A program is a sequence of operations that manipulate states, so we'll add a new construct that performs a sequence of operations. The **do** command evaluates its arguments in order and returns the last argument. It also creates a new "scope" for variables.

```
(do a b c ...)
```

And we add 2 new commands for variable declaration and assignment.

```
(var a init)    ;; create a new variable with a initial value
(set a value)   ;; assign a new value to a variable
```

I've also added comments to the language. A semicolon will skip the rest of the line. Comments are handled by augmenting the **skip_space** function:

```
def skip_space(s, idx):
    while True:
        save = idx
        # try to skip spaces
        while idx < len(s) and s[idx].isspace():
            idx += 1
        # try to skip a line comment
        if idx < len(s) and s[idx] == ';':
            idx += 1
            while idx < len(s) and s[idx] != '\n':
                idx += 1
        # no more spaces or comments
        if idx == save:
            break
    return idx
```

3.3 Variables and Scopes

Variables are scoped — they can only be accessed by their sibling expressions. We'll use a map to store variables while evaluating an expression.

The problem is that subexpressions can define variables whose names collide with a parent scope. This is solved by using per-scope mappings instead of a global mapping.

The **`pl_eval`** function got a new argument (**`env`**) for storing variables.

```
def pl_eval(env, node):
    # read a variable
    if not isinstance(node, list):
        assert isinstance(node, str)
        return name_loopup(env, node)[node]
    ...
```

The **`env`** argument is a linked list, it contains the map for the current scope and the link to the parent scope. Entering and leaving a scope is just adding or removing the list head.

The variable lookup function traverses the list upwards until the name is found.

```
def name_loopup(env, key):
    while env:  # linked list traversal
        current, env = env
        if key in current:
            return current
    raise ValueError('undefined name')
```

The code for evaluating a new scope: create a new map and link it to the current scope. The **`then`** and **`else`** commands are aliases to the **`do`** command. They are just syntactic sugar so that you can write **`(if (then xxx) (else yyy))`** instead of **`(if xxx yyy)`**.

```
def pl_eval(env, node):
    ...
    # new scope
    if node[0] in ('do', 'then', 'else') and len(node) > 1:
        new_env = (dict(), env) # add the map as the linked list head
        for val in node[1:]:
            val = pl_eval(new_env, val)
        return val  # the last item
```

The code for the **`var`** and **`set`** commands is now straightforward.

```
def pl_eval(env, node):
    ...
    # new variable
    if node[0] == 'var' and len(node) == 3:
        _, name, val = node
        scope, _ = env
        if name in scope:
            raise ValueError('duplicated name')
        val = pl_eval(env, val)
        scope[name] = val
        return val
    # update a variable
    if node[0] == 'set' and len(node) == 3:
        _, name, val = node
        scope = name_loopup(env, name)
        val = pl_eval(env, val)
        scope[name] = val
        return val
```

3.4 Testing

The interpreter accepts a sequence of expressions instead of a single expression. This is done by wrapping the input with a **do** command.

```
def pl_parse_prog(s):
    return pl_parse('(do ' + s + ')')
```

A sample program demonstrating variables and scopes:

```
def test_eval():
    def f(s):
        return pl_eval(None, pl_parse_prog(s))

    assert f('''
        ;; first scope
        (var a 1)
```

```
        (var b (+ a 1))
        ;; a=1, b=2
        (do
            ;; new scope
            (var a (+ b 5))     ;; name collision
            (set b (+ a 10))
        )
        ;; a=1, b=17
        (* a b)
    ''') == 17
```

We'll add control flows and functions in the next chapter.

04. Interpreter: Control Flows and Functions

4.1 If-Then-Else

A new command is added for the if-then-else control flow:

```
(if condition yes)
(if condition yes no)
```

The **`if`** command is similar to the **`(? yes no)`** operator in the calculator chapter. Except that the "else" part is optional. So the code is modified to handle both of them.

```
def pl_eval(env, node):
    ...
    # conditional
    if len(node) in (3, 4) and node[0] in ('?', 'if'):
        _, cond, yes, *no = node
        no = no[0] if no else ['val', None] # the `else` part is optional
        new_env = (dict(), env) # new scope
        if pl_eval(new_env, cond):
            return pl_eval(new_env, yes)
        else:
            return pl_eval(new_env, no)
```

Note that I created a new scope before evaluating the condition, this allows a variable declaration in the condition, for example: **`(if (var aaa bbb) (then use aaa here))`**. This is just syntactic sugar.

4.2 Loops

The syntax of the **`loop`** command:

```
(loop condition body)
(break)
(continue)
```

The code for handling the **`loop`** command has one thing in common with the **`if`** command: it translates to a Python loop, just like the **`if`** command translates to a Python **`if`**.

```
def pl_eval(env, node):
    ...
    # loop
    if node[0] == 'loop' and len(node) == 3:
        _, cond, body = node
        ret = None
        while True:
            new_env = (dict(), env)
            if not pl_eval(new_env, cond):
                break
            ret = pl_eval(new_env, body)
        return ret
```

We have also added the **`break`** and **`continue`** commands. They are implemented by (ab)using Python exceptions. You can also propagate them explicitly via the return value of the **`pl_eval`** if you don't like such hacks or you can't use exceptions.

```
def pl_eval(env, node):
    ...
    # loop
    if node[0] == 'loop' and len(node) == 3:
        _, cond, body = node
        ret = None
        while True:
            new_env = (dict(), env)
            if not pl_eval(new_env, cond):
                break
            try:
                ret = pl_eval(new_env, body)
            except LoopBreak:
                break
            except LoopContinue:
                continue
        return ret
    # break & continue
    if node[0] == 'break' and len(node) == 1:
        raise LoopBreak
```

```
    if node[0] == 'continue' and len(node) == 1:
        raise LoopContinue
```

```
class LoopBreak(Exception):
    def __init__(self):
        super().__init__('`break` outside a loop')

class LoopContinue(Exception):
    def __init__(self):
        super().__init__('`continue` outside a loop')
```

4.3 Functions

The syntax:

```
(def func-name (arg-names...) body)
(call func-name args...)
```

The code for function definition does nothing significant. It just performs some sanity checks and puts the function name in the map.

```
def pl_eval(env, node):
    ...
    # function definition
    if node[0] == 'def' and len(node) == 4:
        _, name, args, body = node
        # sanity checks
        for arg_name in args:
            if not isinstance(arg_name, str):
                raise ValueError('bad argument name')
        if len(args) != len(set(args)):
            raise ValueError('duplicated arguments')
        # add the function to the scope
```

```
        dct, _ = env
        key = (name, len(args))
        if key in dct:
            raise ValueError('duplicated function')
        dct[key] = (args, body, env)
        return
```

Note that I added the number of arguments to the key, this allows a form of "overloading" — functions with the same name but different numbers of arguments can coexisit. It also distinguishes function names from variable names.

Now the **`call`** command is handled. Function arguments are treated like new variables. Just create a new scope, put the arguments in it, and evaluate the body.

```
def pl_eval(env, node):
    ...
    # function call
    if node[0] == 'call' and len(node) >= 2:
        _, name, *args = node
        key = (name, len(args))
        fargs, fbody, fenv = name_loopup(env, key)[key]
        # args
        new_env = dict()
        for arg_name, arg_val in zip(fargs, args):
            new_env[arg_name] = pl_eval(env, arg_val)
        # call
        try:
            return pl_eval((new_env, fenv), fbody)
        except FuncReturn as ret:
            return ret.val
```

Special care: the parent scope is not the scope of the caller, but the scope in which the function was defined. (The scope is saved when defining a function).

The **`(return)`** command is handled like the **`(break)`** or **`(continue)`**.

```
def pl_eval(env, node):
    ...
    # return
```

```
    if node[0] == 'return' and len(node) == 1:
        raise FuncReturn(None)
    if node[0] == 'return' and len(node) == 2:
        _, val = node
        raise FuncReturn(pl_eval(env, val))
```

```
class FuncReturn(Exception):
    def __init__(self, val):
        super().__init__('`return` outside a function')
        self.val = val
```

4.4 Done

Congratulations. We have built an interpreter for a mini programming language.

```
def test_eval():
    def f(s):
        return pl_eval(None, pl_parse_prog(s))
    assert f('''
        (def fib (n)
            (if (le n 0)
                (then 0)
                (else (+ n (call fib (- n 1))))))
        (call fib 5)
    ''') == 5 + 4 + 3 + 2 + 1
    assert f('''
        (def fib (n) (do
            (var r 0)
            (loop (gt n 0) (do
                (set r (+ r n))
                (set n (- n 1))
            ))
            (return r)
        ))
```

```
        (call fib 5)
    ''') == 5 + 4 + 3 + 2 + 1
```

Our interpreter is not much more difficult than the calculator in the previous chapter. Variables are solved by extra states, control flows are translated into Python control flows — it's pretty much still simple recursion.

The interpreter still depends on an existing language for execution. How do we compile our language into machine code and run it natively on the CPU? That's the next challenge.

05. How Computers Work

The ultimate goal of this book is to compile our language into machine code that runs natively on the CPU. To do that, we need to have some understanding of how computers work. So let's take a high-level look at what a computer can do, without going into detail.

This chapter is an outline of what you need to know before starting the compiler project. Details will be introduced later.

5.1 Registers, Memory, and Instructions

The CPU takes a list of "instructions" to execute sequentially, which is stored in memory. There are instructions for arithmetic operations, such as adding or multiplying numbers. And there are also instructions for control flows that break sequential execution (branching), such as jumping to another instruction, calling functions, and returning from calls.

A CPU contains a fixed number of internal storage units called "registers". A register stores a certain number of binary bits, often representing an integer, a pointer, or something else (everything is some binary bits). Computations are performed on registers, not directly in memory.

Let's say you want to add up two numbers, the numbers must be loaded from memory into registers, then the CPU can perform the addition. The result of the addition is also stored in a register, which can be transferred back into memory later.

In summary, CPU instructions can be categorized as follows:

1. Arithmetic: Computing on registers.
2. Memory: Load or store between memory and registers.
3. Branching: Moving the execution around.

Note that CPU instructions do not always fall into a single category. In x86 (and x64), arithmetic instructions can also load or store memory. For example, in addition to adding up registers, x86 also allows adding a register to a memory location and vice versa (but not adding up 2 memory locations).

5.2 Function Calls and Stacks

There are usually 2 types of branch instructions:

1. Jump to another instruction based on a condition, this is used for control flows such as if-then-else and loops.
2. Function calls and returns.

Instructions for function calls are not essential. A compiler can use only the first type of branching to implement (emulate) calls and returns. However, there are some conventions for function calls.

When returning from a function call, how does the program know where to go back? A common convention is that before jumping to the target function, the caller pushes its position (the return address) into a stack; thus, the callee knows where to return back by looking at the stack. In x86, there are dedicated instructions that follow this convention: the `call` instruction pushes the current program position into the stack and jumps to the target function, and the `ret` instruction does the reverse.

This is often referred to as the "calling convention". There are other parts of the calling convention, such as how the stack is defined and how arguments are passed, which we'll learn about later.

The stack also stores local variables in addition to the return address. You can see how local variables are "local" — they are removed from the stack when returning from a call.

5.3 System Calls

A program must interact with the operating system for input and output. The mechanism looks just like a normal function call: you call into an operating system routine, and the calling thread suspends until the call returns.

Programming in a high-level programming language such as C or Python rarely invokes system calls directly. The programmer relies on the language runtime or standard library to handle system calls under the hood.

However, since we are building a compiler from scratch, without an existing runtime, we'll work on system calls directly later. A program that does not depend on a runtime or libraries is called a "freestanding" program.

06. The Virtual Machine and Bytecode

6.1 The Two Phases of Compilation

Our compiler project is divided into 2 phases:

1. Compile the syntax tree into instructions of an imaginary virtual machine (VM).
2. Map virtual instructions to x64 instructions and generate Linux ELF executables.

There are many advantages to dividing our compiler into 2 phases, and production-grade compilers can have more than 2 phases. One of the advantages is that our virtual machine is an abstract representation of a real machine; which means it is free of architecture-specific details. The VM can be converted to any real CPU architecture in phase 2 such as x64 or ARM (although we'll only use x64).

The 2-phase design also allows for incremental implementation, which makes the book easier to follow.

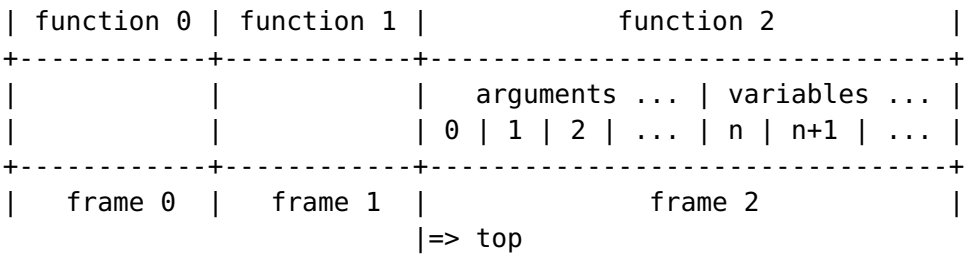
Virtual instructions are widely used in compilers and interpreters and are often called "bytecode". An efficient interpreter executes the program by interpreting the bytecode instead of the syntax tree, although this is not the goal of this book.

6.2 The Virtual Machine Design

There are many possible bytecode designs. But if we design our bytecode to be as close to a real machine as possible, then the conversion to the real machine will be easier. We'll use the following design:

6.2.1 The Data Stack

There is a data stack for local variables and temporary variables. The stack area for a function call is called a "frame"



(The figure above: function 0 calls function 1, which calls function 2).

Variables on the current (top) frame are zero-indexed. Arguments are passed on the frame as if they were local variables. The return value of a function (if it has one) is placed at the position of the first argument.

You may have noticed a difference from real machines: It doesn't store the return address in the stack. How the return address is preserved is an architecture-specific detail that will be handled in phase 2.

6.2.2 Virtual Instructions

Our virtual machine also doesn't have registers, but variables on the stack. Register allocations can be added in phase 2 to map variables to registers. Although this is optional because we can simply load variables into fixed temporary registers for an arithmetic operation and put the result back.

We'll use the following notation for our bytecodes.

- Variables are referred to by their zero-based index on the current frame.
- **`mov src dst`**: Copy the variable **`src`** into the variable **`dst`**.
- **`const val dst`**: Load a constant into the variable **`dst`**.
- **`binop op a1 a2 dst`**: Performs the binary arithmetic operation **`(op a1 a2)`** and stores the result into **`dst`**.
- **`unop op a1 dst`**: For unary operations.
- **`jmpf a1 L`**: Jump to location **`L`** if the value of **`a1`** is false.
- **`jmp L`**: Jump to location **`L`** unconditionally.
- **`call func arg_start level_cur level_new`**: call the function **`func`**. The **`arg_start`** is the index of the first argument (and the return value). The **`level_cur`** and **`level_cur`** are explained later.
- **`ret a1`**: Returns a function call and uses the variable **`a1`** as the return value.
- **`ret -1`**: Return a function without a return value.
- **`syscall dst num args...`**: Invoke the syscall **`num`** and store the return value into **`dst`**.
- **`get_env level var dst`**: Fetch a non-local variable into **`dst`**.
- **`set_env level var src`**: Update a non-local variable with **`src`**.

6.2.3 Non-Local Variables

In many programming languages, functions can access variables besides local variables and arguments. In C, all functions are global and top-level, and variables can be either local or global. Global variables are defined at the top level and can be accessed by functions.

In other languages, such as Python or Go, functions can be nested, a nested function can read and update variables in the parent function, and the language even allows functions to be passed around. This feature is called "closure" — functions that capture variables outside the function and be passed as a value.

There are some extra complications to allowing a nested function to access variables outside of itself. If the variables live on the stack, then it is illegal to call the function that captures them after it has left the scope. This is solved by lifting variables to the heap in GC-enabled languages. Some languages allow you to capture variables by value, which is always accessible, but you cannot update the outside variable by value.

The compiler we're going to implement is none of the above. Our compiler allows nested functions and updating variables outside the function, but has no support for passing functions around; functions are always executed within the scope in which they are defined. This is easier to implement than GC-enabled languages, because we don't need a runtime to support the program. And it is only slightly more complicated than C functions.

6.3 Static Typing

We didn't worry about types when coding the naive interpreter, because we offloaded them to the host language: Python. Almost everything in Python is a heap-allocated "object" that has a type. The Python interpreter constantly checks the type of the objects in order to decide what to do with them. This is called "dynamic typing".

This is in contrast to "static typing", where the compiler knows the type of the values and knows what to do with them at compile time. For example, to execute the expression `a + b` in Python, the interpreter must check the types of both operands, and the execution takes different paths depending on their types: the interpreter may throw an exception, add two numbers, concatenate two strings, and so on.

This is why you don't compile Python into native machine code like you compile C/C++, the compiler doesn't have much work to do as most of the work happens at runtime.

However, there are just-in-time (JIT) compilers that collect type information at runtime and compile Python functions into machine code. This highlights the importance of typing.

We'll use static typing in our language design; it's easier to compile. And it also helps to understand how things work, because the CPU doesn't work with types or objects, the CPU works mostly with integers and pointers!

Our language only supports a few types:

```
int         ;; int64, signed 64-bit integer
byte        ;; uint8, unsigned 8-bit integer
ptr int     ;; pointer to int
ptr byte    ;; pointer to byte
void        ;; only for functions that return nothing
```

07. Compiler: Variables and Expressions

7.1 Introduction

The compiler is divided into small steps. In this chapter, we'll deal with simple expressions and variables. For example, the expression `(+ (- 1 2) 3)` is compiled to the following bytecode sequence.

```
const 1 0       ; load constant 1 into temporary variable #0
const 2 1       ; #1 <- 2
binop - 0 1 0   ; #0 <- #0 - #1
const 3 1       ; #3 <- 1
binop + 0 1 0   ; #0 <- #0 + #1
```

The compiler keeps track of the top of the stack and both local and temporary variables are allocated incrementally on the stack. Local variables always come before temporary variables. Below is a visualization of a single frame with 2 local variables.

```
| var0 | var1 | new variables...
               ^
               |
       == stack top ==>
```

An example with variables: `(var a 1) (+ 3 a)`.

```
const 1 0       ; initialize local variable #0 with constant 1
const 3 1       ; load constant 3 into temporary variable #1
binop + 1 0 1   ; temporary #1 <- #0 + #1
mov 1 0         ; local #0 <- #1, store the result into the variable
```

You may wonder why the **`binop`** instruction doesn't store the result directly into the variable **`a`**. This is because our compiler is naive and does not try to optimize; results are always stored in temporaries.

7.2 Data Structures

Our compiler involves some additional data structures, unlike the simple interpreter we coded before.

7.2.1 class Scope

We'll add the **`Scope`** class for the scope linked list. In addition to the map of variables, it also keeps track of the number of local variables (**`nlocal`**) in that scope. (And **`nlocal`** is currently equal to the size of the **`names`** map, until we add functions in a later chapter).

```
# the name scope linked list
class Scope:
    def __init__(self, prev):
        # the parent scope
        self.prev = prev
        # the number of local variables seen
        self.nlocal = 0
        # Variable names to (type, index) tuples.
        self.names = dict()
```

Looking up a variable name from a scope is the same as before.

```
# lookup a name from a scope. returns a (type, index) tuple.
def scope_get_var(scope, name):
    while scope:
        if name in scope.names:
            return scope.names[name]
        scope = scope.prev
    return None, -1 # not found
```

7.2.2 class Func

The **`Func`** class is the data structure for our compiler. For now, we'll only compile a simple program consisting of a sequence of simple expressions without functions; it is also the data structure for functions in a later chapter.

```
# the compiler state
class Func:
    def __init__(self):
        # the name scope
```

```
        self.scope = Scope(None)
        # the output: a list of instructions
        self.code = []
        # current number of local variable in the stack (non-temporary)
        self.nvar = 0
        # current number of variables (both locals and temporaries)
        self.stack = 0
```

The **scope** field and the **code** field are both obvious.

The **stack** field is the top of the stack. It is used to allocate new variables.

```
class Func:
    # allocate a temporary variable on the stack top and return its index
    def tmp(self):
        dst = self.stack
        self.stack += 1
        return dst
```

The **nvar** field is the number of local variables. It is used to allocate new locals. Local variables always come before temporary variables; so when adding new a local, the stack must be free of temporaries.

```
class Func:
    # allocate a new local variable in the current scope
    def add_var(self, name, tp):
        # add it to the map
        if name in self.scope.names:
            raise ValueError('duplicated name')
        self.scope.names[name] = (tp, self.nvar)    # (type, index)
        self.scope.nlocal += 1
        # assign the index
        assert self.stack == self.nvar
        dst = self.stack
        self.stack += 1
        self.nvar += 1
        return dst
```

You might think that the **nvar** field is redundant, given the assertion:

```
assert self.stack == self.nvar
```

You are right about this, **`nvar`** is also used in several other assertions.

7.3 Compilation Overview

7.3.1 Input and Output

Our compiler does the type inference and bytecode generation in a single pass. Below is the function prototype for compiling an expression. It takes a syntax tree node as the input, generates bytecode (stored in **`Func.code`**), and returns the inferred type and the index of the result variable (if the type is not **`void`**). Every **`pl_comp_*`** function follows this prototype.

```
# returns a (type, index) tuple. the index is -1 if the type is `('void',)`
def pl_comp_xxx(fenv: Func, node):
```

Unlike C or Python, there are no "statements" in our language, everything is an "expression" that has a type. The type can be **`void`** for loops and conditionals, in which case, no variable is allocated and the returned index is -1.

7.3.2 Managing Temporaries

We need to discard temporary variables at some point, after they are no longer useful. We also need to keep temporaries around for a later use in some cases, such as the operands of binary operators and function call arguments. We'll have 2 functions for both cases:

- **`pl_comp_expr`**: Compile and discard temporaries.
- **`pl_comp_expr_tmp`**: Compile and keep temporaries.

Listing the **`pl_comp_expr`** function to show how temporaries are discarded.

```
# the entry point of compilation.
# returns a (type, index) tuple. the index is -1 if the type is `('void',)`
def pl_comp_expr(fenv: Func, node, *, allow_var=False):
    if allow_var:
```

```
        assert fenv.stack == fenv.nvar
    save = fenv.stack

    # the actual implementation
    tp, var = pl_comp_expr_tmp(fenv, node, allow_var=allow_var)
    assert var < fenv.stack

    # Discard temporaries from the above compilation:
    if allow_var:
        # The stack is either local variables only
        fenv.stack = fenv.nvar
    else:
        # or reverts to its previous state.
        fenv.stack = save

    # The result is either a temporary stored at the top of the stack
    # or a local variable.
    assert var <= fenv.stack
    return tp, var
```

We also added the **`allow_var`** flag to control whether new variable declarations are allowed in this expression, e.g., it makes no sense to say **`(+ (var a 1) 2)`**.

7.3.3 Dispatching Commands

The **`pl_comp_expr_tmp`** function simply dispatches the input to other **`pl_comp_*`** functions, each corresponding to a command and following the function prototype mentioned earlier.

```
# The actual implementation of `pl_comp_expr`.
# This preserves temporaries while `pl_comp` discards temporaries.
def pl_comp_expr_tmp(fenv: Func, node, *, allow_var=False):
    # read a variable
    if not isinstance(node, list):
        return pl_comp_getvar(fenv, node)

    # anything else
    if len(node) == 0:
        raise ValueError('empty list')
```

```
    # constant
    if len(node) == 2 and node[0] in ('val', 'val8', 'str'):
        return pl_comp_const(fenv, node)
    # binary operators
    binops = {
        '%', '*', '+', '-', '/',
        'and', 'or',
        'eq', 'ge', 'gt', 'le', 'lt', 'ne',
    }
    if len(node) == 3 and node[0] in binops:
        return pl_comp_binop(fenv, node)
    # unary operators
    if len(node) == 2 and node[0] in {'-', 'not'}:
        return pl_comp_unop(fenv, node)
    # new scope
    if node[0] in ('do', 'then', 'else'):
        return pl_comp_scope(fenv, node)
    # new variable
    if node[0] == 'var' and len(node) == 3:
        if not allow_var:
            # Variable declarations are allowed only as
            # children of scopes and conditions.
            raise ValueError('variable declaration not allowed here')
        return pl_comp_newvar(fenv, node)
    # update a variable
    if node[0] == 'set' and len(node) == 3:
        return pl_comp_setvar(fenv, node)

    raise ValueError('unknown expression')
```

7.4 Local Variables

7.4.1 Variable Declarations

Let's take a look at variables first. The variable declaration command: **`(var name init-value)`**

```
def pl_comp_newvar(fenv: Func, node):
    _, name, kid = node
    # compile the initialization expression
    tp, var = pl_comp_expr(fenv, kid)
    if var < 0: # void
        raise ValueError('bad variable init type')
    # store the initialization value into the new variable
    dst = fenv.add_var(name, tp)
    return tp, move_to(fenv, var, dst)
```

It first compiles the initialization expression, which returns the type of the expression and the index that holds the result (which can be either a temporary or another local variable). The type of the variable is the type of the expression — that's an example of type inference.

Then the **`Func.add_var`** registers the new variable, as explained earlier.

Then it stores the initialization value into the new variable using the **`mov`** bytecode instruction. The bytecode instruction is just a Python tuple.

```
def move_to(fenv, var, dst):
    if dst != var:
        fenv.code.append(('mov', var, dst))
    return dst
```

7.4.2 Variable Lookup and Assignment

The lookup code is simply a bridge to the **`scope_get_var`** function we discussed earlier. This will get more complicated when we add functions and non-local variables later.

```
def pl_comp_getvar(fenv: Func, node):
    assert isinstance(node, str)
    return fenv.get_var(node)

class Func:
    # lookup a name. returns a tuple of (type, index)
    def get_var(self, name):
```

```
        tp, var = scope_get_var(self.scope, name)
        if var >= 0:
            return tp, var
        raise ValueError('undefined name')
```

Assignment: **`(set name expr)`**

```
def pl_comp_setvar(fenv: Func, node):
    _, name, kid = node

    dst_tp, dst = fenv.get_var(name)
    tp, var = pl_comp_expr(fenv, kid)
    if dst_tp != tp:
        raise ValueError('bad variable set type')

    return dst_tp, move_to(fenv, var, dst)
```

7.4.3 Scopes

The **`(do a b c...)`** command creates a new scope, it is also the only place where variable declarations are allowed (so far). It returns the last subexpression.

```
def pl_comp_scope(fenv: Func, node):
    fenv.scope_enter()
    tp, var = ('void',), -1
    for kid in node[1:]:
        tp, var = pl_comp_expr(fenv, kid, allow_var=True)
    fenv.scope_leave()

    # the return is either a local variable or a new temporary
    if var >= fenv.stack:
        var = move_to(fenv, var, fenv.tmp())
    return tp, var
```

Entering a new scope is simply adding a node to the scope linked list. Leaving a scope is the opposite. Note the stack cleanup here.

```
class Func:
    # enter a new scope
    def scope_enter(self):
        self.scope = Scope(self.scope)  # new list head
        self.scope.save = self.stack

    # exit a scope and revert the stack
    def scope_leave(self):
        self.stack = self.scope.save
        self.nvar -= self.scope.nlocal
        self.scope = self.scope.prev
```

After the stack cleanup, the result may refer to a discarded variable, in which case, we need to move the value to a new temporary.

7.5 Operators

The **`binop`** bytecode instruction is used for all binary operators. There are 2 variants for the type **`int`** and type **`byte`**. Pointers will be added in a later chapter.

```
binop  op arg1 arg2 dst     ; stores `(op arg1 arg2)` into `dst`
binop8 op arg1 arg2 dst     ; same but for the `byte` type
```

The compilation of its operands uses **`pl_comp_expr_tmp`** instead of the **`pl_comp_expr`**, the former doesn't clean up the stack so that the second operand doesn't overwrite the first operand.

```
def pl_comp_binop(fenv: Func, node):
    op, lhs, rhs = node

    # compile subexpressions
    # FIXME: boolean short circuit
    save = fenv.stack
    t1, a1 = pl_comp_expr_tmp(fenv, lhs)
    t2, a2 = pl_comp_expr_tmp(fenv, rhs)
    fenv.stack = save   # discard temporaries
```

```
    # TODO: pointers
    if 'ptr' in (t1[0], t2[0]):
        raise NotImplementedError

    # check types
    if not (t1 == t2 and t1[0] in ('int', 'byte')):
        raise ValueError('bad binop types')
    rtype = t1
    if op in {'eq', 'ge', 'gt', 'le', 'lt', 'ne'}:
        rtype = ('int',)    # boolean

    suffix = ''
    if t1 == t2 and t1 == ('byte',):
        suffix = '8'
    # output to a new temporary
    dst = fenv.tmp()
    fenv.code.append(('binop' + suffix, op, a1, a2, dst))
    return rtype, dst
```

The function also checks the type of the 2 operands and infers the result type. The **`int`** type is used for comparison operators as we don't have a dedicated boolean type.

The **`unop`** and **`unop8`** are for unary operators and should be obvious at this point, so we'll skip the code listing.

We'll add control flows such as loops and if-then-else in the next chapter.

08. Compiler: Control Flows

8.1 Introduction

In this chapter, we'll add if-then-else and loop control flows, both of which utilize the branching bytecode instructions **`jmp`** and **`jmpf`**.

Consider a simple example:

```
(var a 10)
(loop (gt a 0)
    (set a (- a 1)))
```

Which is equivalent to:

```
a = 10
while a > 0:
    a = a - 1
```

The bytecode sequence for this is:

```
        const 10 0
    L0:
        const 0 1
        binop gt 0 1 1
        jmpf 1 L1
        const 1 1
        binop - 0 1 1
        mov 1 0
        jmp L0
    L1:
```

In the above listing, the labels **`L0`** and **`L1`** denote 2 locations in the bytecode sequence, they are the destinations of the **`jmp`** and **`jmpf`** instructions. The **`jmpf 1 L1`** breaks the loop if the condition is false and the **`jmp L0`** closes the loop.

From the above example, you can see that control flows are just a bunch of conditional jumps and unconditional jumps; you can always rewrite loops and if-then-else in this way if your language supports the **`goto`** statement. This is also how real computers work.

8.2 If-Then-Else

Let's start with the **`if`** command. In addition to conditionally executing the **`then`** and the **`else`** subexpressions, it also returns the value of the executed subexpression if both are of the same type, otherwise, the return type is **`void`**.

```
(if condition yes)
(if condition yes no)
```

8.2.1 Labels

Labels are used to keep track of the destination of the **`jmp`** instruction. They are numeric IDs paired with their locations. You may wonder why we don't use locations directly in the **`jmp`** instruction. This is because when we emit a forward jump, the destination location hasn't been compiled yet; it's convenient to add a level of indirection and handle it later.

```
class Func:
    def __init__(self):
        ...
        # label IDs to instruction locations
        self.labels = []

    # allocate a new label ID
    def new_label(self):
        l = len(self.labels)
        self.labels.append(None)    # filled later
        return l

    # associate the label ID to the current location
    def set_label(self, l):
        assert l < len(self.labels)
        self.labels[l] = len(self.code)
```

8.2.2 Conditionals

The bytecode sequence for the **`if`** command follows this pattern:

```
    <condition>
    jmpf idx L0     ; skip the <then> to the <else>
    <then>
    jmp L1          ; skip the <else>
L0:
    <else>
L1:
```

The code listing for the **`if`** command:

```
def pl_comp_cond(fenv: Func, node):
    _, cond, yes, *no = node
    l_true = fenv.new_label()   # then
    l_false = fenv.new_label()  # else
    fenv.scope_enter()  # a variable declaration is allowed on the condition

    # the condition expression
    tp, var = pl_comp_expr(fenv, cond, allow_var=True)
    if tp == ('void',):
        raise ValueError('expect boolean condition')
    fenv.code.append(('jmpf', var, l_false))    # go to `else` if false

    # then
    t1, a1 = pl_comp_expr(fenv, yes)
    if a1 >= 0:
        # Both `then` and `else` goes to the same variable,
        # thus a temporary is needed.
        move_to(fenv, a1, fenv.stack)

    # else, optional
    t2, a2 = ('void',), -1
    if no:
        fenv.code.append(('jmp', l_true))   # skip `else` after `then`
    fenv.set_label(l_false)
    if no:
        t2, a2 = pl_comp_expr(fenv, no[0])
```

```
        if a2 >= 0:
            move_to(fenv, a2, fenv.stack)   # the same variable for `then`
    fenv.set_label(l_true)

    fenv.scope_leave()
    if a1 < 0 or a2 < 0 or t1 != t2:
        return ('void',), -1    # different types, no return value
    else:
        return t1, fenv.tmp()   # allocate the temporary for the result
```

There are 2 additional things that need further explanation:

- The **`mov`** instruction after evaluating each subexpression: Both results are moved to the same temporary so that we can return either of them (if their types match).
- The **`allow_var=True`** and the new scope: This is an optional syntax sugar mentioned in the interpreter chapter.

8.3 Loops

The pattern for loops is probably trivial to you at this point:

```
L0:
    <condition>
    jmpf idx L1
    <body>
    jmp L0
L1:
```

The code listing for the **`loop`** command:

```
def pl_comp_loop(fenv: Func, node):
    _, cond, body = node
    fenv.scope.loop_start = fenv.new_label()
    fenv.scope.loop_end = fenv.new_label()

    # enter
    fenv.scope_enter()  # allow_var=True
    fenv.set_label(fenv.scope.loop_start)
```

```
    # cond
    _, var = pl_comp_expr(fenv, cond, allow_var=True)
    if var < 0: # void
        raise ValueError('bad condition type')
    fenv.code.append(('jmpf', var, fenv.scope.loop_end))
    # body
    _, _ = pl_comp_expr(fenv, body)
    # loop
    fenv.code.append(('jmp', fenv.scope.loop_start))
    # leave
    fenv.set_label(fenv.scope.loop_end)
    fenv.scope_leave()

    return ('void',), -1
```

One more thing to explain: the labels for entering and leaving a loop are stored in the **Scope** class.

```
# the name scope linked list
class Scope:
    def __init__(self, prev):
        # the parent scope
        self.prev = prev
        ...
        # the label IDs of the nearest loop
        self.loop_start = prev.loop_start if prev else -1
        self.loop_end = prev.loop_end if prev else -1
```

It is used to implement the **break** and **continue** commands. Note that child scopes inherit loop labels from their parent so that you can **break** and **continue** anywhere inside a loop.

```
def pl_comp_expr_tmp(fenv: Func, node, *, allow_var=False):
    ...
    # break & continue
    if node == ['break']:
        if fenv.scope.loop_end < 0:
            raise ValueError('`break` outside a loop')
        fenv.code.append(('jmp', fenv.scope.loop_end))
```

```
        return ('void'), -1
    if node == ['continue']:
        if fenv.scope.loop_start < 0:
            raise ValueError('`continue` outside a loop')
        fenv.code.append(('jmp', fenv.scope.loop_start))
        return ('void'), -1
    ...
```

You can try to add the **`goto`** command as an exercise, which is more generic than **`break`** and **`continue`** and is common in C-like languages. Beware of the constraints regarding variables and scopes, though.

09. Compiler: Functions and Non-Locals

9.1 Introduction

The syntax for defining and calling functions:

```
(def (func-name return-type) (arg-name-type-pairs...) body)
(call func-name args...)
```

A few examples:

```
;; no arguments and no return value
(def (f void) () (do a b c))
(call f)
;; returns a int and takes 2 int as argument
(def (g int) ((arg1 int) (arg2 int)) (do a b c))
(call g 1 2)
```

We have to explicitly define the return type and the argument types. The **`void`** type can be used for functions without a return value.

Since functions can be nested, variables are either local or non-local, and our language has no concept of global variables. Below is a sample program that demonstrates non-local variables. See the "Virtual Machine" chapter for the language design.

```
(def (f int) ((x int)) (do
    (var a x)
    (def (inc void) ((b int)) (set a (+ a b)))  ;; update `a`
    (def (dec void) ((b int)) (set a (- a b)))
    (call inc 5)
    (call dec 2)
    (return a)
))
```

9.2 Function Definitions

9.2.1 Data Structures

The **`Func`** class is modified to compile functions, and the program itself is contained in a special function **`main`**, which is the first function to be compiled.

```
# the compiler state for functions
class Func:
    def __init__(self, prev):
        # the parent function (linked list)
        self.prev = prev
        # nested function level. the level of `main` is 1.
        self.level = (prev.level + 1) if prev else 0
        # the return type of this function
        self.rtype = None
        # a list of all functions. shared by all functions in a program.
        self.funcs = prev.funcs if prev else []
        ...
```

4 new fields are added:

1. **`Func.prev`**: Since functions can be nested, the **`Func`** object is now a linked list node pointing to its parent function, thus enabling non-local variable lookups.
2. **`Func.level`**: Nested function level; explained later.
3. **`Func.rtype`**: The return type, used for type-checking.
4. **`Func.funcs`**: A collection of all functions. Each function has its own bytecode output.

The program is wrapped within the **`main`** function.

```
def pl_parse_main(s):
    return pl_parse('(def (main int) () (do ' + s + '))')
```

9.2.2 Function Definitions

The function compilation is divided into 2 functions. The first function is **`pl_scan_func`**, it does 3 things:

1. Validate the return type and argument types (**`validate_type`**).
2. Add the function to the **`Scope.names`** map. The rule for looking up functions is the same as for variables, so the map stores both variable names and function names. And you can see why **`Scope.nlocal`** is necessary for counting local variables. Also note that the map key includes argument types to allow function overloading.
3. Create a new **`Func`** object and add it to the **`Func.funcs`** collection. Its index in **`Func.funcs`** is stored in the **`Scope.names`** map.

```
# function preprocessing:
# make the function visible to the whole scope before its definition.
def pl_scan_func(fenv: Func, node):
    _, (name, *rtype), args, _ = node
    rtype = validate_type(rtype)

    # add the (name, arg-types) pair to the map
    arg_type_list = tuple(validate_type(arg_type) for _, *arg_type in args)
    key = (name, arg_type_list) # allows overloading by argument types
    if key in fenv.scope.names:
        raise ValueError('duplicated function')
    fenv.scope.names[key] = (rtype, len(fenv.funcs))

    # the new function
    func = Func(fenv)
    func.rtype = rtype
    fenv.funcs.append(func)
    return func
```

The second function is **`pl_comp_func`**, which actually compiles the function into bytecode. Arguments are handled as if they were local variables, and there is the **`ret`** bytecode instruction at the end.

```
# actually compile the function definition.
# note that the `fenv` argument is the target function!
def pl_comp_func(fenv: Func, node):
    _, _, args, body = node

    # treat arguments as local variables
    for arg_name, *arg_type in args:
        if not isinstance(arg_name, str):
```

```
            raise ValueError('bad argument name')
        arg_type = validate_type(arg_type)
        if arg_type == ('void',):
            raise ValueError('bad argument type')
        fenv.add_var(arg_name, arg_type)
    assert fenv.stack == len(args)

    # compile the function body
    body_type, var = pl_comp_expr(fenv, body)
    if fenv.rtype != ('void',) and fenv.rtype != body_type:
        raise ValueError('bad body type')
    if fenv.rtype == ('void',):
        var = -1
    fenv.code.append(('ret', var))  # the implicit return
    return ('void',), -1
```

Here is how the **main** function is compiled. It shows that the 2 functions are used together.

```
def pl_comp_main(fenv: Func, node):
    assert node[:3] == ['def', ['main', 'int'], []]
    func = pl_scan_func(fenv, node)
    return pl_comp_func(func, node)
```

9.2.3 Mutual Function Calls

You may wonder why we split the function routine into 2 phases. This is to allow 2 functions to call each other mutually. Let's say that the function **foo** is defined before the function **bar**, and the **foo** wants to call the **bar** and vice versa. If we compile them sequentially, the **foo** cannot find the **bar** since it is not compiled yet.

The solution is to add functions to the scope map before compiling them. This is handled in the **do** command, not in the **pl_comp_expr** function, and it is the only place where functions are allowed.

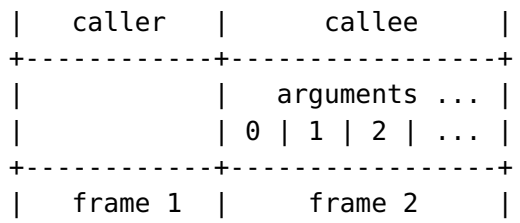
```
def pl_comp_scope(fenv: Func, node):
    fenv.scope_enter()
    tp, var = ('void',), -1

    # split kids into groups separated by variable declarations
    groups = [[]]
    for kid in node[1:]:
        groups[-1].append(kid)
        if kid[0] == 'var':
            groups.append([])

    # Functions are visible before they are defined,
    # as long as they don't cross a variable declaration.
    # This allows adjacent functions to call each other mutually.
    for g in groups:
        # preprocess functions
        funcs = [
            pl_scan_func(fenv, kid)
            for kid in g if kid[0] == 'def' and len(kid) == 4
        ]
        # compile subexpressions
        for kid in g:
            if kid[0] == 'def' and len(kid) == 4:
                target, *funcs = funcs
                tp, var = pl_comp_func(target, kid)
            else:
                tp, var = pl_comp_expr(fenv, kid, allow_var=True)

    fenv.scope_leave()

    # the return is either a local variable or a new temporary
    if var >= fenv.stack:
        var = move_to(fenv, var, fenv.tmp())
    return tp, var
```

Note that we have divided subexpressions into groups separated by variable declarations, and only functions within a group can see each other before they are compiled. This is to prevent uninitialized variable access.

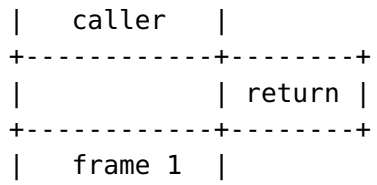
9.3 Function Calls and Returns

Arguments are placed on the stack in an orderly fashion before invoking a function call. The callee then treats the arguments as predefined local variables, and places the return value in the variable of the first argument.

After the call:



After the return:



Code listing:

```
def pl_comp_call(fenv: Func, node):
    _, name, *args = node

    # compile arguments
    arg_types = []
    for kid in args:
        tp, var = pl_comp_expr(fenv, kid)
        arg_types.append(tp)
        move_to(fenv, var, fenv.tmp())  # stored continuously
    fenv.stack -= len(args) # points to the first argument

    # look up the target `Func`
    key = (name, tuple(arg_types))
    _, _, idx = fenv.get_var(key)
    func = fenv.funcs[idx]

    fenv.code.append(('call', idx, fenv.stack, fenv.level, func.level))
    dst = -1
    if func.rtype != ('void',):
```

```
        dst = fenv.tmp()    # the return value on the stack top
    return func.rtype, dst
```

The **`call`** bytecode instruction takes several operands:

1. **`idx`**: The index of the target function.
2. **`fenv.stack`**: The first argument on the stack. This is also the start of the new frame.
3. **`fenv.level`** and **`func.level`**: These are used for machine code generation in a later chapter.

The **`(return)`** command takes an optional return value (if the return type is not **`void`**) and emits a **`ret`** bytecode instruction.

```
def pl_comp_return(fenv: Func, node):
    _, *kid = node
    tp, var = ('void',), -1
    if kid:
        tp, var = pl_comp_expr_tmp(fenv, kid[0])
    if tp != fenv.rtype:
        raise ValueError('bad return type')
    fenv.code.append(('ret', var))
    return tp, var
```

9.4 Non-Local Variables

(See the "Virtual Machine" chapter for the design.)

The variable lookup code is modified to continue the search on the parent function, this is necessary for both non-local variables and function lookups. It also returns the nested level of the scope so that we can distinguish local variables from non-local variables.

```
class Func:
    # lookup a name. returns a tuple of (function_level, type, index)
    def get_var(self, name):
        tp, var = scope_get_var(self.scope, name)
        if var >= 0:
```

```
            return self.level, tp, var
        if not self.prev:
            raise ValueError('undefined name')
        return self.prev.get_var(name)
```

To access a non-local variable in the parent function, the start position of the parent frame must be known. But in our virtual machine design, the stack only allows direct access to the current frame, and operators only work with locals; non-locals are read via the **get_env** bytecode instruction, which copies a non-local variable into a local variable. How the program finds the non-local variable is deferred to the machine code phase.

```
def pl_comp_getvar(fenv: Func, node):
    assert isinstance(node, str)
    flevel, tp, var = fenv.get_var(node)
    if flevel == fenv.level:
        # local variable
        return tp, var
    else:
        # non-local
        dst = fenv.tmp()
        fenv.code.append(('get_env', flevel, var, dst))
        return tp, dst
```

Likewise, updating a non-local variable is compiled into the **set_env** bytecode instruction.

```
def pl_comp_setvar(fenv: Func, node):
    _, name, kid = node

    flevel, dst_tp, dst = fenv.get_var(name)
    tp, var = pl_comp_expr(fenv, kid)
    if dst_tp != tp:
        raise ValueError('bad variable set type')

    if flevel == fenv.level:
        # local
        return dst_tp, move_to(fenv, var, dst)
    else:
```

```
        # non-local
        fenv.code.append(('set_env', flevel, dst, var))
        return dst_tp, move_to(fenv, var, fenv.tmp())
```

9.5 Closing Remarks

Our bytecode compiler is mostly finished at this point. The next phase is to actually run the bytecode.

You can build an interpreter by interpreting the bytecode. This is often more efficient than the naive interpreter in earlier chapters. This is because the bytecode compilation removes many aspects of the naive interpreter, such as variable lookups and type checking. However, this is not the direction of the rest of the book.

The next phase is translating our bytecode into machine code, and generating native binary executables, which requires some machine-specific and operating-system-specific knowledge. We'll cover these first.

10. A Crash Course on x64 Assembly

Some basics on x64 assembly and the instruction format are useful to our compiler. This is not a complete description of x64 assembly, and you will need to consult references and manuals in later chapters.

You may want to review the "How Computers Work" chapter while reading this chapter.

10.1 Registers

There are 16 64-bit general-purpose registers. Registers are referred to by their names in assembly languages.

```
rax rcx rdx rbx
rsp rbp rsi rdi
r8  r9  r10 r11
r12 r13 r14 r15
```

An x64 instruction takes a maximum of 2 operands. Let's take the **`add`** instruction as an example:

```
add rax, rcx    ; comment: rax = rax + rcx
```

The first operand stores the output of the arithmetic, in this example it's also one of the inputs. Note that there are other assembly syntaxes that use a different order of operands, which may confuse you.

There are also 32-bit, 16-bit, and 8-bit registers. They are not independent registers but are part of a 64-bit register. The 32-bit **`eax`** register is the lower 32-bit of the **`rax`** register. Changing the **`eax`** register affects **`rax`** and vice versa. Likewise, the **`ax`** and **`al`** registers are the lower 16-bit and 8-bit of the **`rax`** register, respectively.

Below is the table of register names, note the pattern of naming.

```
 64  32  16   8
rax eax  ax  al
rcx ecx  cx  cl
rdx edx  dx  dl
rbx ebx  bx  bl
rsp esp  sp spl
rbp ebp  bp bpl
r8  r8d r8w r8l
r9  r9d r9w r9l
...
```

There is a special effect regarding 32-bit registers: the upper 32-bit of a 64-bit register is automatically zeroed when the lower 32-bit is changed via the 32-bit register. This is unlike 16-bit and 8-bit registers, where the upper bits remain intact when the 16-bit or 8-bit register is changed.

10.2 Addressing Modes

In x64, arithmetic instructions can also perform a memory load or store. Let's use the **add** instruction again as an example:

```
; rcx contains a memory address, load an integer from that address,
; then add that integer to rax.
; In C: rax = rax + *rcx;   // rcx is a pointer
add rax, [rcx]

; Load an integer from the address in rcx,
; then add it with rax,
; then store the result back to the address in rcx.
; In C: *rcx = *rcx + rax
add [rcx], rax
```

Only one of the operands can refer to memory. Instructions like **add [rax], [rcx]** do not exist! This is explained later.

There are several ways to refer to memory besides the example above.

```
add rax, [rcx]              ; mentioned before
add rax, [rcx + 16]         ; register + displacement
add rax, [rdx + 8*rbx]      ; base + scale * index
add rax, [rdx + 8*rbx + 16] ; base + scale * index + displacement
```

The memory address is calculated from the formula inside the `[]`. This is not a complete list of addressing modes, and there are additional restrictions on these formulas, such as the `scale` can only be 1, 2, 4, or 8.

In many x64 instructions, either of the operands can refer to memory using the above addressing modes. And there is a special instruction that does nothing but calculate the memory address from an addressing mode.

```
; compute the address from the 2nd operand and store it to a register
lea rax, [rdx + 8*rbx]  ; rax = rdx + 8*rbx
```

`lea` = Load Effective Address.

10.3 The mov Instruction

The `mov` instruction has multiple functions.

```
mov rax, 123    ; load a constant (immediate) into rax
mov rax, rbx    ; copy rbx into rax
mov rax, [rbx]  ; load an integer into rax from memory address [rbx]
mov [rbx], rax  ; store rax into memory
```

At first glance, the 4 examples seem to be unrelated. Loading constants into registers, moving stuff between registers, memory loads and stores, what do they have in common?

If you consider the addressing modes in the previous section, then only the first line (loading a constant) is new to you, the rest 3 lines are just using addressing modes. The function of the last 3 lines is to copy an operand to another operand.

The first line `mov rax, 123` copies a constant into the destination operand, the constant is encoded in the instruction and is called "immediate" in the manuals. Some arithmetic instructions also take this form, such as `add rax, 123`.

And the destination operand can also use addressing modes:

```
; copy a 64-bit integer constant to address [rax]
mov qword ptr [rax], 123    ; In C: *rax = 123
; add a constant to a 64-bit integer to address [rax]
add qword ptr [rax], 123    ; In C: *rax = *rax + 123
```

Note that the **qword ptr** is needed before the **[rax]**. It is used to specify the size of the operand, because the assembler has no idea what size the constant 123 is, so you cannot just say **mov [rax], 123**. The **dword ptr**, **word ptr**, and **byte ptr** are for 32-bit, 16-bit and 8-bit operands respectively.

10.4 Control Flows

The **rip** register is a special register that always contains the address of the next instruction. Unlike the 16 general-purpose registers, **rip** cannot be read or updated directly, nor can it be used in arithmetic instructions. The instructions to change **rip** are used to implement control flows.

There is also another addressing mode that refers to a memory location relative to **rip**:

```
mov rax, [rip + 8]  ; load an integer from the address [rip + 8]
```

10.4.1 Unconditional Jumps

The **jmp** instruction changes **rip** unconditionally. And like other instructions you've seen, it can take several different forms, each with a different function.

One of the forms is the relative jump, which takes a constant (immediate) as the relative offset to the current **rip**. The offset can be either positive or negative, which corresponds to jumping forward or backward. Unfortunately, you cannot write out this form directly in assembly languages; the assembler takes a "label" as the destination and automatically fills in the offset.

```
label_a:
    blah blah
    ...
    jmp label_a ; rip = <address_at_label_a>
```

Another form is the absolute jump, which takes a register or memory operand as the new **rip**. See examples:

```
jmp rax     ; rip = rax
jmp [rax]   ; rip = *rax
```

10.4.2 Conditional Jumps

Let's start with an example:

```
cmp rax, rcx    ; compare rax and rcx
jl L            ; jump to L if rax < rcx, assuming signed integers
```

The **jl** instruction stands for "jump if less", there are many other conditional jump instructions:

```
jl      ; jump if less, signed integer
jb      ; jump if below, unsigned integer
jnb     ; jupm if not below, unsigned integer
ja      ; jump if above, unsigned integer
jae     ; jump if above or equal, unsigned integer
je      ; jump if equal
jne     ; jump if not equal
...
```

The naming pattern is obvious, you can use any combination of the following letters if it makes sense. This group of instructions is referred to as **jcc** in the manuals, where **cc** stands for "condition code".

```
n: not
e: equal
a: above, unsigned
b: below, unsigned
g: greater, signed
l: less, signed
```

Note that this is not a complete list, there are more letters! And **`jnb`** is apparently the same instruction as **`jae`**.

(If you are wondering about the distinction between signed and unsigned integers, you must first learn binary numbers and "two's complement" before you can read this chapter.)

10.4.3 Flags

The CPU must store the result of the **`cmp`** instruction somewhere, so that a following conditional jump instruction can use it. The result is stored in the **`FLAGS`** register, which consists of various bits called "flags". An incomplete list of the flags:

```
CF  Carry flag
ZF  Zero flag
SF  Sign flag
OF  Overflow flag
```

These flags are set to 0 or 1 based on the result of the computation. For the **`cmp`** instruction, the computation is the subtraction of the 2 operands; the instruction **`cmp rax, rcx`** sets flags based on the result of **`rax - rcx`**:

- **`CF`**: Assuming they are unsigned integers, the flag is set to 1 if **`rax - rcx`** is below zero (implies **`rax < rcx`**), which cannot be represented by an unsigned integer, otherwise, the flag is set to 0. The **`jb`** (jump if below) instruction checks this flag.
- **`ZF`**: Set to 1 if the result is zero, otherwise 0. **`je`** and **`jne`** both check this flag.
- **`SF`**: Set to the most significant bit of the result, which is the sign bit for signed integers.
- **`OF`**: Like **`CF`**, but for signed integers.

With these flags, both signed and unsigned integer comparisons can be represented. Some conditional jumps check for multiple flags, for example, **`jbe`** checks both **`CF`** and **`ZF`**. To further your understanding, you can try to answer the question: Which flag(s) does **`jg`** check?

The **`cmp`** instruction is not the only instruction that sets flags, the **`sub`** instruction is for subtracting operands, which is the same calculation as in **`cmp`**, and it sets flags in the same way. The only difference is that **`cmp`** discards the result, while **`sub`** overwrites the first operand with the result. Other arithmetic instructions such as **`add`**, **`and`**, and etc. also set flags.

The **test** instruction is also for comparisons like **cmp**, it uses bitwise **and** instead of **sub**. Here is a common way to test whether a value is zero or not:

```
test rax, rax   ; compute rax & rax
je L            ; jump if rax == 0, implied by (rax & rax) => rax => 0
```

test is also used to test whether certain bits are set, hence the name.

10.5 Function Calls and the Stack

10.5.1 Push and Pop

The **rsp** register has a special function, it can be used for stack-related operations. There are dedicated instructions for manipulating stacks: the **push** and **pop** instructions. It is assumed that **rsp** points to the stack top and the stack grows downward (to the lower address).

The **push** instruction *decreases* **rsp** by 8, and puts a 64-bit value to the location where **rsp** points to:

```
push 123    ; 64-bit immediate. In C: *(--rsp) = 123;
push rax    ; value from a register
push [rax]  ; load the value from memory
```

The **pop** instruction is the opposite, it fetches a 64-bit value into the operand and *increases* **rsp** by 8.

```
pop rax     ; In C: rax = *rsp++;
pop [rax]   ; In C: *rax = *rsp++;
```

10.5.2 Call and Return

For function calls, the stack can be used to pass arguments and store local variables, but most importantly, the stack is used to store the return address!

To initiate a function call, the **`call`** instruction pushes **`rip`** (which points to the next instruction) into the stack and jumps to the destination function. It's very much like the **`jmp`** instruction, except for the stack push.

To return from a call, the **`ret`** instruction pops the return address from the stack into **`rip`**, which points to the next instruction of **`call`**.

There are other details of function calls, such as how arguments and the return value are passed, which are part of the "calling conventions" that must be followed when calling external functions compiled by C/C++.

10.5.3 System Calls

The instruction for invoking a system call is **`syscall`**, it's similar to **`call`**, except that it takes a number instead of an address. The operating system intercepts the syscall and performs the operation corresponding to the syscall number. This is similar to a normal function call from the program's point of view.

The system call arguments are only passed in registers according to the calling convention. To invoke a syscall, **`rax`** is set to the syscall number, and the arguments are put in 6 registers in the following order:

```
rdi rsi rdx r10 r8 r9
```

The return value or **`errno`** is in **`rax`** after the return. For example, to invoke the **`read`** Linux syscall, which has the following prototype in C:

```
ssize_t read(int fd, char *buf, size_t count);
```

Registers:

- **`rax`**: The syscall number, which is 0 in this case.
- **`rdi`**: The **`fd`** argument.
- **`rsi`**: The **`buf`** argument.
- **`rdx`**: The **`count`** argument.
- **`r10`**, **`r8`**, **`r9`**: Not used.

The calling convention for syscalls limits the number of syscall arguments to a maximum value of 6, and arguments must fit into a register (integers or pointers).

11. The x64 Instruction Encoding

Since we are going to generate binary executables, we need to know how to encode x64 instructions into binary bits. And this is a "from scratch" project, so we won't be relying on an external assembler to convert assembly text.

Learning the instruction encoding will also help you understand the x64 assembly better. For example, there are some mysterious constraints, such as why you can't do something in assembly, like **`add [rax], [rcx]`** or **`mov rax, rip`**, that will be clarified once you dive into the encoding.

11.1 Instruction Format

All x64 instructions conform to the following format:

```
| prefix | opcode | ModR/M | SIB | displacement | immediate |
|        | 1,2,3B |   1B   |  1B |   1,2,4,8B   |  1,2,4,8B |
```

The **`opcode`** field is present in all instructions, the presence of other fields is determined by the previous fields. Let's take the **`add rax, 0x12`** instruction as an example, it is encoded into 4 bytes:

```
add rax, 0x12
+-----------------------------------+
|   48   |   83   |   c0   |   12   |
| prefix | opcode | ModR/M | immediate |
```

The prefix and opcode together (**`0x48`**, **`0x83`**) determine that this is an **`add`** instruction containing the ModR/M field and a 1-byte immediate. The ModR/M field represents the operand register **`rax`** in this case. The immediate byte **`0x12`** is encoded as is. Details are explained later.

If you look up the manual, you'll find that there are multiple opcodes for the **`add`** instruction. For example, **`add rax, rcx`** encodes into 3 bytes:

```
add rax, rcx
+-------------------------+
|   48   |   03   |   c1   |
| prefix | opcode | ModR/M |
```

The opcode is **0x03** in this case, which is different from the previous example — it distinguishes itself from other forms of the **add** instruction. You can think that **add** in assembly languages is not a single instruction, but a common notation for a group of instructions.

The ModR/M byte **0xc1** represents both register operands (**rax**, **rcx**).

Let's examine 3 more examples, with the same opcode **0x03**:

```
add rax, [rcx]
+-------------------------+
|   48   |   03   |   01   |
| prefix | opcode | ModR/M |

add rax, [rcx + 0x12]
+---------------------------------------+
|   48   |   03   |   41   |      12      |
| prefix | opcode | ModR/M | displacement |

add rax, [rcx + 8*rbx + 0x12]
+-----------------------------------------------+
|   48   |   03   |   44   |  d9 |      12      |
| prefix | opcode | ModR/M | SIB | displacement |
```

The ModR/M byte can encode not only register operands but also memory address modes. It determines which addressing modes to use, which determines whether the SIB byte and the displacement field are present. SIB stands for "scale-index-base", it's an additional byte for some addressing modes because the information does not fit into the 1-byte ModR/M field.

11.2 ModR/M Byte

The ModR/M byte consists of the following bit fields:

```
  7                           0
+---+---+---+---+---+---+---+---+
|  mod  |     reg     |     rm    |
+---+---+---+---+---+---+---+---+
```

The 3-bit field **reg** encodes a register. The **rm** field can also encode a memory addressing mode instead of a register.

Remember that there are 16 general-purpose registers, we need 4 bits instead of 3 to encode them. There are 2 additional bits in the prefix byte, each extending **reg** and **rm** to 4 bits. This detail is explained later.

What addressing modes does the instruction use? This is determined by the 2-bit **`mod`** field:

1. If **`mod`** is **`0b11`**, then **`rm`** refers to a plain register, just like **`reg`**, and the displacement field is not present.
2. If **`mod`** is **`0b00`**, then **`rm`** uses addressing mode **`[rm]`**, and the displacement field is not present.
3. If **`mod`** is **`0b01`**, then **`rm`** uses addressing mode **`[rm + disp8]`**. **`disp8`** means a 1-byte displacement field.
4. If **`mod`** is **`0b10`**, then **`rm`** uses addressing mode **`[rm + disp32]`**. **`disp32`** means a 4-byte displacement field.

There are exceptions when **`mod`** is not **`0b11`**:

1. If **`rm`** is **`rsp`** or **`r12`**, then the "scale-index-base" addressing mode is used. A SIB byte is used to encode the addressing mode instead. And **`mod`** determines the presence and the size of the displacement field using the same rule as before, which is **`[SIB]`**, **`[SIB + disp8]`**, or **`[SIB + disp32]`**.
2. If **`rm`** is **`rbp`** or **`r13`**, and **`mod`** is **`0b00`**, then the **`[rip + disp32]`** addressing mode is used.

Let's review some of the previous examples:

```
add rax, rcx
+----------------------------------+
|   48   |   03   |        c1       |
| prefix | opcode |      ModR/M     |
                  | mod | reg |  rm |
                    11   000   001
                         rax   rcx

add rax, [rcx]
+----------------------------------+
|   48   |   03   |        01       |
| prefix | opcode |      ModR/M     |
                  | mod | reg |  rm |
                    00   000   001
                         rax   rcx

add rax, [rcx + 0x12]
+-------------------------------------------------+
|   48   |   03   |        41       |      12      |
| prefix | opcode |      ModR/M     | displacement |
                  | mod | reg |  rm |
                    01   000   001
                         rax   rcx
```

11.3 SIB Byte

The SIB byte encodes the addressing modes **`[base + scale * index]`**:

```
  7                           0
+---+---+---+---+---+---+---+---+
| scale |   index   |    base   |
+---+---+---+---+---+---+---+---+
```

- **`scale`** is a 2-bit field. It refers to 1, 2, 4, or 8.
- **`base`** refers to a general-purpose register. Like the **`rm`** field, it is extended to 4 bits by the prefix.
- **`index`** is also a register.

The **`mod`** field determines the displacement field in the same way as before.

There is also an exception for **`rsp`**: it cannot be the index register. If **`rsp`** is encoded as **`index`**, the addressing mode becomes just **`[base]`**. So an address like **`[rsp + rsp]`** does not exist.

The **`rbp`** and **`r13`** is another exception, they cannot be the base register if **`mod`** is **`0b00`**.

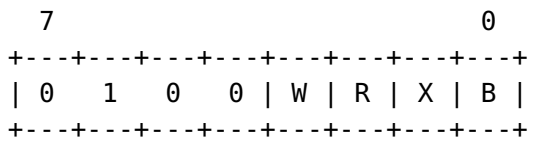
If both of the 2 exceptions are met, there is neither base nor index, and the addressing mode becomes just **`[disp32]`**.

Some examples:

```
add rax, [rcx + 8*rbx + 0x12]
+-------------------------------------------------------------+
|   48   |   03   |        44       |       d9       |   12  |
| prefix | opcode |      ModR/M     |      SIB       | disp8 |
                  | mod | reg |  rm | sc | idx | bas |
                  |  01   000   100 | 11   011   001 |
                         rax   rsp    8   rbx   rcx

add rax, [rsp + 0x12]
+-------------------------------------------------------------+
|   48   |   03   |        44       |       24       |   12  |
| prefix | opcode |      ModR/M     |      SIB       | disp8 |
                  | mod | reg |  rm | sc | idx | bas |
                  |  01   000   100 | 00   100   100 |
                         rax   rsp    1   rsp   rsp
```

11.4 Prefixes

So far, all the examples use the prefix **`0x48`**, this is the so-called REX prefix.



- The REX prefix starts with a fixed bit pattern **`0100`** in the upper 4 bits.
- The **`W`** bit determines whether the operands are 64-bit or 32-bit, hence all the examples above use the REX prefix with **`W`** set to 1.
- The **`R`** bit is combined with the 3-bit **`reg`** field to encode 16 general-purpose registers. Thus, the REX prefix is required for instructions using **`r8`** to **`r15`**.
- The **`X`** bit is combined with the 3-bit **`index`** field.
- The **`B`** bit is combined with either **`rm`** or **`base`**.

The x64 instruction set is an extension to x86. In x86, there are only 8 general-purpose registers (**`eax`** to **`edi`**), so 3 bits are enough to encode a register. In x64, the number of general-purpose registers doubles, so the REX prefix is designed to add more bits.

See some examples below. Note the change of the prefix.

```
add rax, rax  ; R reg => 0 000, B rm => 0 000
+-------------------------------------+
|     48    |   03   |        c0      |
|    REX    | opcode |      ModR/M    |
| 0100 WRXB |        | mod | reg |  rm |
       1000             11   000   000
                             rax   rax

add rax, r8   ; R reg => 0 000, B rm => 1 000
+-------------------------------------+
|     49    |   03   |        c0      |
|    REX    | opcode |      ModR/M    |
| 0100 WRXB |        | mod | reg |  rm |
       1001             11   000   000
                            0000  1000
                             rax    r8
```

```
add eax, r8d  ; W=0, 32-bit operand size
+--------------------------------------+
|     41    |   03   |        c0       |
|    REX    | opcode |      ModR/M     |
| 0100 WRXB |        | mod | reg |  rm |
       0001             11   000   000
                            0000  1000
                             eax   r8d

add eax, eax  ; no REX prefix
+-------------------------+
|   03   |        c0      |
| opcode |      ModR/M    |
|        | mod | reg |  rm |
           11   000   000
                eax   eax
```

There are other prefixes as well. The **`0x66`** prefix changes the operand size to 16-bit.

```
add ax, ax    ; 0x66 prefix, 16-bit operand size
+-------------------------------------+
|     66    |   03   |        c0      |
|   prefix  | opcode |      ModR/M    |
|           |        | mod | reg |  rm |
                        11   000   000
                              ax    ax

add ax, r8w   ; 2 prefixes
+-----------------------------------------+
| 66 |     41    |   03   |        c0      |
|    |    REX    | opcode |      ModR/M    |
|    | 0100 WRXB |        | mod | reg |  rm |
           0001             11   000   000
                                 0000  1000
                                   ax   r8w
```

12. From Virtual Machine To Real Machine

With the knowledge of x64 assembly, we can start translating our bytecode into machine code.

12.1 Stacks

Let's review what we learned about stacks in the previous chapters. - The `rsp` points to the stack top, and it grows *downwards*. - The `call` instruction pushes the return address into the stack and jumps to the target. - The `ret` instruction pops the return address into `rip`.

This is what we'll use for function calls. We also need to store local variables in the stack. However, *we'll use a different design*: besides the call stack `rsp`, we also use an extra data stack for local variables. The data stack top is pointed to by `rbx` and grows *upwards*. This is just an arbitrary design, most compilers just use the `rsp` stack for both calls and local data.

The `rbx` stack stores both named and temporary local variables. Since we don't have many data types, all variables in the data stack use exactly 8 bytes. And this stack grows upwards, so the nth variable is addressed as `[rbx + 8*n]`.

When an operation is performed on a variable, the operands are first loaded into registers, and the result is stored back into the variable in the stack. The stack always holds up-to-date values.

We need temporary registers for the result of arithmetic and operands, binary operators take 2 operands so we only need 2 registers. We'll use `rax` and `rdx` for the operands, and `rax` for the result.

There are 16 general-purpose registers on x64, and most of them are usable for holding values, but we'll only use `rax` and `rdx`. This is a naive way to generate machine code and doesn't make full use of the hardware; an optimizing compiler will try to map variables to registers to reduce unnecessary loads and stores. But for now, we'll start with the simple and naive way.

12.2 CodeGen class

The **CodeGen** class is for machine code generation. It contains a list of register encodings and a few mappings that are explained later.

```
class CodeGen:
    # register encodings
    A = 0
    C = 1
    D = 2
    B = 3
    SP = 4
    BP = 5
    SI = 6
    DI = 7

    def __init__(self):
        # output
        self.buf = bytearray()
        # states
        self.jmps = dict()      # label -> offset list
        self.calls = dict()     # function index -> offset list
        self.strings = dict()   # string literal -> offset list
        self.func2off = []      # func idx -> offset
```

A program consists of functions, and a function consists of bytecode instructions. To translate a function to machine code, each bytecode instruction is translated into machine code one by one. For each bytecode instruction, there is a method of the same name in the **CodeGen** class.

```
class CodeGen:
    # compile a function
    def func(self, func: Func):
        # call the method for each instruction
        for instr_name, *instr_args in func.code:
            method = getattr(self.__class__, instr_name)
            method(self, *instr_args)
```

12.3 Load and Store

Let's start with a simple instruction: the **mov** bytecode instruction which copies a variable into another variable.

```
class CodeGen:
    def mov(self, src, dst):
        if src == dst:
            return
        self.load_rax(src)
        self.store_rax(dst)
```

It loads the source into **rax** and stores **rax** back into the target. Both the load and store are mapped to the **mov** x64 instruction.

```
class CodeGen:
    # mov reg, [rm + disp]
    def asm_load(self, reg, rm, disp):
        self.asm_disp(b'\x48\x8b', reg, rm, disp)

    # mov [rm + disp], reg
    def asm_store(self, rm, disp, reg):
        self.asm_disp(b'\x48\x89', reg, rm, disp)

    def store_rax(self, dst):
        # mov [rbx + dst*8], rax
        self.asm_store(CodeGen.B, dst * 8, CodeGen.A)

    def load_rax(self, src):
        # mov rax, [rbx + src*8]
        self.asm_load(CodeGen.A, CodeGen.B, src * 8)
```

The **asm_disp** function is used to generate x64 instructions in the format of **prefix opcode ModR/M displacement** and the **rm** field refers to the addressing mode **[rm + disp]**. Both the load **mov** and the store **mov** conform to this format, and they are distinguished by different opcodes (**0x8b** vs. **0x89**).

Depending on the displacement value, instructions are encoded with 0, 1, or 4 displacement bytes.

```
class CodeGen:
    # instr reg, [rm + disp]
    # instr [rm + disp], reg
    def asm_disp(self, lead, reg, rm, disp):
        assert reg < 16 and rm < 16 and rm != CodeGen.SP

        lead = bytearray(lead)  # optional prefix + opcode
        if reg >= 8 or rm >= 8:
            assert (lead[0] >> 4) == 0b0100 # REX
            lead[0] |= (reg >> 3) << 2      # REX.R
            lead[0] |= (rm >> 3) << 0       # REX.B
            reg &= 0b111
            rm &= 0b111

        self.buf.extend(lead)
        if disp == 0:
            mod = 0     # [rm]
        elif -128 <= disp < 128:
            mod = 1     # [rm + disp8]
        else:
            mod = 2     # [rm + disp32]
        self.buf.append((mod << 6) | (reg << 3) | rm)  # ModR/M
        if mod == 1:
            self.i8(disp)
        if mod == 2:
            self.i32(disp)

    # append a signed integer
    def i8(self, i):
        self.buf.append(i if i >= 0 else (256 + i))
    def i32(self, i):
        self.buf.extend(struct.pack('<i', i))
    def i64(self, i):
        self.buf.extend(struct.pack('<q', i))
```

This is a direct application of the instruction format in the previous chapter.

12.4 Constants

Continuing with another instruction: the **`const`** instruction which loads a constant into a variable.

```
class CodeGen:
    def const(self, val, dst):
        assert isinstance(val, (int, str))
        if isinstance(val, str):
            # lea rax, [rip + offset]
            self.buf.extend(b"\x48\x8d\x05")
            self.strings.setdefault(val, []).append(len(self.buf))
            self.buf.extend(b"\0\0\0\0")    # placeholder
        else:
            # integers
            ...
        self.store_rax(dst)
```

It handles both strings and integers. String literals in our language are just pointers to null-terminated constant strings, as in C. Constant strings are stored somewhere in the memory along with the code, so they can be addressed by the **`rip`** relative addressing **`[rip + disp32]`**. But, we don't know the offset when translating this instruction, so we'll just use a placeholder displacement value **`\0\0\0\0`** and record the location that refers to this string.

The placeholder displacement is filled in later when string literals are appended to the end of the output.

```
class CodeGen:
    # fill in a 4-byte `rip` relative offset
    def patch_addr(self, patch_off, dst_off):
        src_off = patch_off + 4     # rip
        relative = struct.pack('<i', dst_off - src_off)
        self.buf[patch_off:patch_off+4] = relative

    def code_end(self):
        ...
        # strings
        for s, off_list in self.strings.items():
```

```
            dst_off = len(self.buf)
            for patch_off in off_list:
                self.patch_addr(patch_off, dst_off)
            self.buf.extend(s.encode('utf-8') + b'\0')
        self.strings.clear()
```

Integers are handled differently because the constants can be encoded as the immediate value of the **mov** instruction; no need to load them from memory.

```
class CodeGen:
    def const(self, val, dst):
        assert isinstance(val, (int, str))
        if isinstance(val, str):
            ...
        elif val == 0:
            self.buf.extend(b"\x31\xc0")            # xor eax, eax
        else:
            self.buf.extend(b"\x48\xb8")            # mov rax, imm64
            self.i64(val)
        self.store_rax(dst)
```

Note that I have added a shortcut: **xor eax, eax**. It's the best way to set a register to 0, and it's shorter than **mov** instructions. Also, note that it's **eax** instead of **rax**; the upper 32 bits are automatically cleared when updating the 32-bit register, which saves us a byte by skipping the REX prefix.

There are other shortcuts for loading constants: the **mov** instructions support 32-bit immediate, which can be used when the constant is small enough. This is left as an exercise for the reader. Be aware of the difference between the zero-extend and the sign-extend, though.

12.5 Control Flows

The **jmp** and **call** instructions are similar, both of them are in the form of **opcode disp32**. And like the **rip** relative addressing for string literals, they just leave behind placeholders.

```
class CodeGen:
    def jmp(self, L):
        self.buf.extend(b"\xe9")    # jmp
        self.jmps.setdefault(L, []).append(len(self.buf))
        self.buf.extend(b'\0\0\0\0')

    def jmpf(self, a1, L):
        self.load_rax(a1)
        self.buf.extend(
            b"\x48\x85\xc0"         # test rax, rax
            b"\x0f\x84"             # je
        )
        self.jmps.setdefault(L, []).append(len(self.buf))
        self.buf.extend(b'\0\0\0\0')

    def asm_call(self, L):
        self.buf.extend(b"\xe8")    # call
        self.calls.setdefault(L, []).append(len(self.buf))
        self.buf.extend(b'\0\0\0\0')
```

When translating bytecode, the location of the corresponding machine code is recorded, so that the displacement values of **`jmp`** can be determined in the end.

```
class CodeGen:
    def func(self, func: Func):
        # offsets
        self.func2off.append(len(self.buf)) # function index -> code offset
        pos2off = []    # virtual instruction -> code offset

        # call the method for each instruction
        for instr_name, *instr_args in func.code:
            pos2off.append(len(self.buf))
            method = getattr(self.__class__, instr_name)
            method(self, *instr_args)

        # fill in the jmp address
        for L, off_list in self.jmps.items():
            dst_off = pos2off[func.labels[L]]
            for patch_off in off_list:
```

```
                self.patch_addr(patch_off, dst_off)
        self.jmps.clear()
```

The **`call`** offsets are handled similarly after all functions have been translated.

```
class CodeGen:
    def code_end(self):
        # fill in the call address
        for L, off_list in self.calls.items():
            dst_off = self.func2off[L]
            for patch_off in off_list:
                self.patch_addr(patch_off, dst_off)
        self.calls.clear()
        # strings
        ...
```

12.6 Function Calls

12.6.1 Calls and Returns

The **`call`** bytecode instruction must maintain the pointer pointing to the stack top. **`rbx`** points to the first argument before jumping to the target function, and it reverts to the previous value after the call is returned.

```
class CodeGen:
    def call(self, func, arg_start, level_cur, level_new):
        ...
        # make a new frame and call the target
        self.buf.extend(b"\x48\x81\xc3")    # add rbx, arg_start*8
        self.i32(arg_start * 8)
        self.asm_call(func)                 # call func
        self.buf.extend(b"\x48\x81\xc3")    # add rbx, -arg_start*8
        self.i32(-arg_start * 8)
        ...
```

Arguments and the return value are passed on the data stack, this does not follow any calling conventions as we are not interacting with foreign functions.

```
class CodeGen:
    def ret(self, a1):
        if a1 > 0:
            self.load_rax(a1)
            self.store_rax(0)
        self.buf.append(0xc3)       # ret
```

12.6.2 Non-Local Variables

There are some additional complications caused by non-local variables: How do we know the location of a non-local variable? Our virtual machine design leaves out the details to the **`get_env`** and **`set_env`** bytecode instructions.

The local variables in the current frame are accessed via **`rbx`**, but in order to access variables in an outer frame, we need to keep track of the location of the target frame. This is done by putting a list of pointers to every frame of nested functions in the **`rsp`** stack. When entering a function call, the list could grow its size by one (by calling a nested function), keep its size, or reduce its size. And after returning from a function call, the list in the stack is discarded.

The **`level_cur`** and **`level_new`** are the function levels of the caller and the callee; they are used to determine the size of the list.

```
class CodeGen:
    def call(self, func, arg_start, level_cur, level_new):
        assert 1 <= level_cur
        assert 1 <= level_new <= level_cur + 1

        # put a list of pointers to outer frames in the `rsp` stack
        if level_new > level_cur:
            # grow the list by one
            self.buf.append(0x53)               # push rbx
        for _ in range(min(level_new, level_cur) - 1):
            # copy the previous list
```

```
            self.buf.extend(b"\xff\xb4\x24")    # push [rsp + (level_new-1)*8]
            self.i32((level_new - 1) * 8)

        # make a new frame and call the target
        ...

        # discard the list of pointers
        self.buf.extend(b"\x48\x81\xc4")        # add rsp, (level_new - 1)*8
        self.i32((level_new - 1) * 8)
```

To access a non-local variable, first load the target frame pointer from the **rsp** stack into **rax**, then use the pointer in the same way as **rbx**.

```
class CodeGen:
    def load_env_addr(self, level_var):
        self.buf.extend(b"\x48\x8b\x84\x24")    # mov rax, [rsp + level_var*8]
        self.i32(level_var * 8)

    def get_env(self, level_var, var, dst):
        self.load_env_addr(level_var)
        # mov rax, [rax + var*8]
        self.asm_load(CodeGen.A, CodeGen.A, var * 8)
        # mov [rbx + dst*8], rax
        self.store_rax(dst)

    def set_env(self, level_var, var, src):
        self.load_env_addr(level_var)
        # mov rdx, [rbx + src*8]
        self.asm_load(CodeGen.D, CodeGen.B, src * 8)
        # mov [rax + var*8], rdx
        self.asm_store(CodeGen.A, var * 8, CodeGen.D)
```

12.7 Operators

Let's first look at some binary operators. One of the operands is loaded into **rax** while the other one is used as a memory operand, the result is stored back into memory from **rax**.

```
class CodeGen:
    def binop(self, op, a1, a2, dst):
        self.load_rax(a1)
        arith = {
            '+': b'\x48\x03',       # add  reg, rm
            '-': b'\x48\x2b',       # sub  reg, rm
            '*': b'\x48\x0f\xaf',   # imul reg, rm
        }
        if op in arith:
            # op rax, [rbx + a2*8]
            self.asm_disp(arith[op], CodeGen.A, CodeGen.B, a2 * 8)
        ...
        self.store_rax(dst)
```

The **idiv** instruction is for both the division and modulo of signed integers. It's different from other arithmetic instructions: The first operand (dividend) isn't a 64-bit register you can choose freely; it's a 128-bit integer taken from a fixed pair of registers **rdx:rax**. **rdx** is upper and **rax** is lower. The result is a 64-bit quotient stored in **rax** and a 64-bit remainder stored in **rdx**.

```
class CodeGen:
    def binop(self, op, a1, a2, dst):
        self.load_rax(a1)
        ...
        if op in ('/', '%'):
            # xor edx, edx
            self.buf.extend(b"\x31\xd2")
            # idiv rax, [rbx + a2*8]
            self.buf.extend(b'\x48\xf7\xbb')
            self.i32(a2 * 8)
            if op == '%':
                # mov rax, rdx
                self.buf.extend(b"\x48\x89\xd0")
        ...
        self.store_rax(dst)
```

Here, we introduce a new group of instructions: **setcc**. The **cc** stands for "condition code", which is the same as in the conditional jump instructions **jcc**. It's used to store the boolean result of comparison into an 8-bit register.

```
class CodeGen:
    def binop(self, op, a1, a2, dst):
        self.load_rax(a1)
        ...
        cmp = {
            'eq': b'\x0f\x94\xc0',  # sete  al
            'ne': b'\x0f\x95\xc0',  # setne al
            'ge': b'\x0f\x9d\xc0',  # setge al
            'gt': b'\x0f\x9f\xc0',  # setg  al
            'le': b'\x0f\x9e\xc0',  # setle al
            'lt': b'\x0f\x9c\xc0',  # setl  al
        }
        if op in cmp:
            # cmp rax, [rbx + a2*8]
            self.asm_disp(b'\x48\x3b', CodeGen.A, CodeGen.B, a2 * 8)
            # setcc al
            self.buf.extend(cmp[op])
            # movzx eax, al
            self.buf.extend(b"\x0f\xb6\xc0")
        ...
        self.store_rax(dst)
```

Note that we need to clear the upper bits when dealing with 8-bit registers. **`movzx`** is "move with zero-extend", which pads the source operand with zeros.

12.8 Executing Machine Code

The machine code generation is mostly done. The next step is to dump machine code into an executable file, which will be done in a later chapter. For now, we can test our compiler by simply executing machine code from memory, which is also used by JIT compilers.

12.8.1 The Wrapper Function

To execute a piece of machine code in Python, the machine code must be wrapped into a C function, which must follow the calling conventions for C code. Any C function can be invoked using the **`ctypes`** module.

Since our program uses an additional data stack, let's allocate it in Python and pass it as an argument. And the function should also return the return value of **main**. The wrapper function prototype is:

```
int64_t wrapper(void *stack);
```

The calling conventions are part of the "System V AMD64 ABI", it says that the first 6 arguments are passed in registers in the following order:

```
rdi, rsi, rdx, rcx, r8, r9
```

And the result is returned in **rax**. Below is the wrapper function that follows the calling conventions.

```
class CodeGen:
    # C function: int64_t (*)(void *stack)
    def mem_entry(self):
        # the first argument is the data stack
        self.buf.extend(b"\x53")            # push rbx
        self.buf.extend(b"\x48\x89\xFB")    # mov rbx, rdi
        # call the main function
        self.asm_call(0)
        # the return value
        self.buf.extend(b"\x48\x8b\x03")    # mov rax, [rbx]
        self.buf.extend(b"\x5b")            # pop rbx
        self.buf.extend(b"\xc3")            # ret
```

The calling conventions also say that some registers must be preserved through function calls, which is usually done by saving and restoring affected registers with **push** and **pop**:

```
rbx, rsp, rbp, r12, r13, r14, r15
```

Put all the bits together:

```
class CodeGen:
    # compile to a callable function
    def output_mem(self, root: Func):
        self.mem_entry()
        for func in root.funcs:
            self.func(func)
        self.code_end()
```

12.8.2 mmap and ctypes

The application memory is normally not executable, and there are 3 bits that control whether a memory page is readable, writable, or executable. To execute machine code, we need to allocate a chunk of executable memory using the **mmap** syscall, which is part of the Python stdlib. The relevant bit is **PROT_EXEC**.

```
# execute the program as a ctype function
class MemProgram:
    def __init__(self, code):
        # copy the code to an executable memory buffer
        flags = mmap.MAP_PRIVATE|mmap.MAP_ANONYMOUS
        prot = mmap.PROT_EXEC|mmap.PROT_READ|mmap.PROT_WRITE
        self.code = mmap.mmap(-1, len(code), flags=flags, prot=prot)
        self.code[:] = code

        # ctype function: int64_t (*)(void *stack)
        func_type = ctypes.CFUNCTYPE(ctypes.c_int64, ctypes.c_void_p)
        cbuf = ctypes.c_void_p.from_buffer(self.code)
        self.cfunc = func_type(ctypes.addressof(cbuf))
        ...
```

After copying the machine code into the executable memory, the **ctypes** wrapper is created.

- **ctypes.CFUNCTYPE** is used to declare the prototype.
- **ctypes.c_void_p.from_buffer** creates a Python object representing a **void** *. Its content is the address of the underlying **mmap** buffer.

- **`ctypes.addressof`** retrieves the content of this **`void *`**.

Our program also needs an additional data stack, which is also conveniently created via **`mmap`**.

```
# execute the program as a ctype function
class MemProgram:
    def __init__(self, code):
        ...
        # create the data stack
        prot = mmap.PROT_READ|mmap.PROT_WRITE
        self.stack = mmap.mmap(-1, 8 << 20, flags=flags, prot=prot)
        cbuf = ctypes.c_void_p.from_buffer(self.stack)
        self.stack_addr = ctypes.addressof(cbuf)

    def invoke(self):
        return self.cfunc(self.stack_addr)
```

Let's hook everything up:

```
def main():
    ...

    # source text
    with open(args.file, 'rt', encoding='utf-8') as fp:
        text = fp.read()

    # parse & compile
    node = pl_parse_main(text)
    root = Func(None)
    _ = pl_comp_main(root, node)

    # execute
    gen = CodeGen()
    gen.output_mem(root)
    prog = MemProgram(gen.buf)
    sys.exit(prog.invoke())
```

12.8.3 Testing

Let's run a test program that returns 42 as the return code.

```
$ cat samples/42.txt
(def (foo int) ((n int))
    (if (le n 0)
        (then 1)
        (else (* n (call foo (- n 1))))))
(/ (call foo 7) (call foo 5))
$ ./pl_comp_x64.py --exec ./samples/42.txt
$ echo $?
42
```

To run a classic hello world, we need to add the **`syscall`** command and bytecode. The calling conventions for syscalls are different, as described in the assembly chapter.

```
class CodeGen:
    def syscall(self, dst, num, *arg_list):
        self.buf.extend(b"\xb8")                # mov eax, imm32
        self.i32(num)
        arg_regs = [CodeGen.DI, CodeGen.SI, CodeGen.D, 10, 8, 9]
        assert len(arg_list) <= len(arg_regs)
        for i, arg in enumerate(arg_list):
            # mov reg, [rbx + arg*8]
            self.asm_load(arg_regs[i], CodeGen.B, arg * 8)
        self.buf.extend(b"\x0f\x05")            # syscall
        self.store_rax(dst)                     # mov [rbx + dst*8], rax
```

Now we can run a classic hello world:

```
;; the write() syscall:
;; ssize_t write(int fd, const void *buf, size_t count);
(syscall 1 1 "Hello world!\n" 13)
0
```

13. Binary Executables

In this chapter, we'll dump machine code into a freestanding executable file.

13.1 The ELF Format

The Executable and Linkable Format (ELF) is the standard binary executable file format for Unix and Unix-like systems. At the highest level, an ELF executable is roughly a container for machine code.

13.1.1 The Program Header

To execute a piece of machine code, the code must be loaded into the memory. This is the job of the so-called "program header" in ELF — it maps a portion of the file into memory at a specific virtual address, as if the file were **`mmap`**ed.

A program header contains:

- The virtual address where the mapping starts.
- The size of the mapping.
- The offset in the ELF file.

There can be multiple program headers and they create multiple memory mappings, possibly with different access modes. The mapping for machine code can be read and executed, but not written. In addition, an ELF may have a read-only and non-executable mapping for constants, and/or a read-write mapping for global variables. Access modes are also specified in program headers.

13.1.2 The Section Header

In addition to program headers, there are also "section headers". *Section headers are not required for ELF executables*. They can be used to define different roles of data sections (machine code, constants, or global variables), which can be used by ELF manipulation tools such as linkers and dissemblers. For example, the machine code section is named ".text", when you run the **`objdump -d`** to dissemble an executable, it looks for the ".text" section.

A section header contains:

- The name of the section.
- The size of the section.
- The offset in the ELF file.

13.1.3 The ELF Header

Where do all these headers go, and how many of them? This is defined in the "ELF header" at the beginning of the file.

The ELF header contains:

- The number of program headers.
- The offset of the first program header. All program headers are placed together.
- The number of section headers.
- The offset of the first section header. Section headers are also placed together.
- The "entry point".

The entry point is the virtual address where the machine code execution starts, and it is usually inside the ".text" section.

13.1.4 The File Layout

So far, we have enough information to read an ELF file. What about creating an ELF file, where to put these sections and headers? The only requirement for the layout is that the ELF header is at the beginning, everything else can be placed anywhere, as long as it can be reached by offsets.

We'll use this layout: ELF header + program header + code. And since section headers are optional, we'll ignore them entirely.

13.2 Creating ELF Files

13.2.1 Headers and Structures

The process is broken down into 3 steps.

```
class CodeGen:
    # compile the program to an ELF executable
    def output_elf(self, root: Func):
        # ELF header + program header
        self.elf_begin()
        # machine code
        self.code_entry()
        for func in root.funcs:
            self.func(func)
        self.code_end()
        # fill in some ELF fields
        self.elf_end()
```

The first step is to generate headers.

```
class CodeGen:
    def elf_begin(self):
        self.elf_header()
        ...

    def elf_header(self):
        self.buf.extend(bytes.fromhex('7F 45 4C 46 02 01 01 00'))
        self.buf.extend(bytes.fromhex('00 00 00 00 00 00 00 00'))
        # e_type, e_machine, e_version
        self.buf.extend(bytes.fromhex('02 00 3E 00 01 00 00 00'))
        self.f64('e_entry')
        self.f64('e_phoff')
        self.f64('e_shoff')
        self.f32('e_flags')
        self.f16('e_ehsize')
        self.f16('e_phentsize')
        self.f16('e_phnum')
        self.f16('e_shentsize')
        self.f16('e_shnum')
        self.f16('e_shstrndx')
        self.setf('e_phoff', len(self.buf))     # offset of the program header
        self.setf('e_ehsize', len(self.buf))    # size of the ELF header
```

As you can see, there are a lot of fields, but some of them are irrelevant to our purpose; they are set to fixed values or zeros. The relevant fields will be filled in later. In this case, we

can fill in the **e_ehsize** field (ELF header size) and the **e_phoff** field (offset to the program header) right after the ELF header is written.

Some helper functions for manipulating fields are added:

```
class CodeGen:
    def __init__(self):
        ...
        self.fields = dict()    # ELF field name -> (size, offset)

    # append a placeholder field
    def f16(self, name):
        self.fields[name] = (2, len(self.buf))
        self.buf.extend(b'\0\0')
    def f32(self, name):
        ...
    def f64(self, name):
        ...

    # fill in the placeholder
    def setf(self, name, i):
        sz, off = self.fields[name]
        fmt = {2: '<H', 4: '<I', 8: '<Q'}[sz]
        self.buf[off:off+sz] = struct.pack(fmt, i)
```

The ELF header is followed by a program header. We're also filling a few fields with fixed values here.

- The **p_vaddr** field is the virtual address used to map the executable file, its value doesn't matter here since our compiled machine code doesn't depend on absolute addresses.
- The **p_offset** field is the start of the mapping. For simplicity, we'll just map the whole file into memory, hence it's 0.

```
class CodeGen:
    def __init__(self):
        # params
        self.vaddr = 0x1000     # the virtual address for the program
        ...
```

```
    def elf_begin(self):
        self.elf_header()

        phdr_start = len(self.buf)  # the program header starts here
        self.elf_program_header()
        # program header size
        self.setf('e_phentsize', len(self.buf) - phdr_start)
        # number of program headers: 1
        self.setf('e_phnum', 1)
        ...

    def elf_program_header(self):
        # p_type, p_flags
        self.buf.extend(bytes.fromhex('01 00 00 00 05 00 00 00'))
        # p_offset
        self.i64(0)
        # p_vaddr, p_paddr
        self.i64(self.vaddr)
        self.i64(self.vaddr)    # useless
        self.f64('p_filesz')
        self.f64('p_memsz')
        # p_align
        self.i64(0x1000)
```

The only important field is **`e_entry`**, it's the virtual address where the machine code starts. Since the mapping starts at the beginning of the file, the virtual address after headers is calculated as "virtual address + offset".

```
class CodeGen:
    def elf_begin(self):
        ...
        self.padding()
        # the entry point: the virtual address where the program start
        self.setf('e_entry', self.vaddr + len(self.buf))
```

The **`CodeGen.padding`** pads the output to be 16-byte-aligned, this just makes it easier to find things in hexdump.

The size of the mapping (in the program header) is filled in last.

```
class CodeGen:
    def elf_end(self):
        # fields in program header:
        # the size of the mapping. we're mapping the whole file here.
        self.setf('p_filesz', len(self.buf))
        self.setf('p_memsz', len(self.buf))
```

13.2.2 The Entry Point

The **code_entry** function generates the startup code that calls the **main** function. The **e_entry** field points to it and the program starts from it. It's more complicated than the **mem_entry** we used before to execute the program from memory.

1. Our program needs an additional data stack, which is created by invoking the **mmap** syscall with assembly.
2. In the end, we need to invoke the **exit** syscall with the return code from **main**.

```
class CodeGen:
   def code_entry(self):
        # create the data stack
        self.buf.extend(
            b"\xb8\x09\x00\x00\x00"     # mov eax, 9
            b"\x31\xff"                 # xor edi, edi      // addr = NULL
            b"\x48\xc7\xc6"             # mov rsi, xxx      // len
        )
        self.i32(8 << 20)   # 8M
        self.buf.extend(
            # prot = PROT_READ|PROT_WRITE
            b"\xba\x03\x00\x00\x00"     # mov edx, 3
            # flags = MAP_PRIVATE|MAP_ANONYMOUS
            b"\x41\xba\x22\x00\x00\x00" # mov r10d, 0x22
            b"\x49\x83\xc8\xff"         # or r8, -1         // fd = -1
            b"\x4d\x31\xc9"             # xor r9, r9        // offset = 0
            b"\x0f\x05"                 # syscall
            b"\x48\x89\xc3"             # mov rbx, rax      // the data stack
            # FIXME: check the syscall return value
        )
```

```
        # call the main function
        self.asm_call(0)
        # exit
        self.buf.extend(
            b"\xb8\x3c\x00\x00\x00"     # mov eax, 60
            b"\x48\x8b\x3b"             # mov rdi, [rbx]
            b"\x0f\x05"                 # syscall
        )
```

13.3 Mission Completed

The last thing to do is to write the output to a file.

```
def main():
    ...
    # parse & compile
    node = pl_parse_main(text)
    root = Func(None)
    _ = pl_comp_main(root, node)
    if args.print_ir:
        print(ir_dump(root))

    # generate output
    if args.output:
        gen = CodeGen()
        gen.output_elf(root)
        fd = os.open(args.output, os.O_WRONLY|os.O_CREAT|os.O_TRUNC, 0o755)
        with os.fdopen(fd, 'wb', closefd=True) as fp:
            fp.write(gen.buf)
```

The ELF file can be disassembled with the **`objdump -D`** command:

```
objdump -b binary -M intel,x86-64 -m i386 \
    --adjust-vma=0x1000 --start-address=0x1080 \
    -D $ELF_FILE
```

`0x1000` is the address where the file is mapped to virtual memory. And the entry point is **`0x1080`**, which skips the headers. These arguments are required because we didn't use a section header named ".text" to tell **`objdump`** where to disassemble.

14. Pointers and Memory

Although we have produced working executables from the compiler, the language itself isn't very interesting. We can only perform calculations on integers and print out fixed message strings, we cannot even print out integers.

The missing part is pointers. With pointers added, we can manipulate strings and perform dynamic memory allocations.

14.1 New Commands

The **`peek`** and **`poke`** commands are used to dereference and write to pointers. And there are new bytecode instructions corresponding to them.

```
(peek ptr)
(poke ptr value)
```

Pointers can be created by referencing variable names.

```
(ref name)
```

Or they can be obtained from the **`mmap`** syscall, which dynamically allocates a piece of variable-sized memory. The **`(syscall num ...)`** command returns an **`int`** that must be cast to a pointer type. We'll add a new command for typecastings

```
(cast type val)
```

We'll also add the **`(ptr elem_type)`** command, which returns a null pointer of the type **`(ptr elem_type)`**. It's a shortcut to **`(cast (ptr elem_type) 0)`**.

14.2 Demos

Below is how we allocate memory dynamically via **`mmap`**.

```
(def (mmap ptr byte) ((n int)) (do
    (var prot 3)            ;; PROT_READ|PROT_WRITE
    (var flags 0x22)        ;; MAP_PRIVATE|MAP_ANONYMOUS
    (var fd -1)
    (var offset 0)
    (var r (syscall 9 0 n prot flags fd offset))
    (return (cast (ptr byte) r))
))
```

The prototype of the **mmap** syscall:

```
void *mmap(
   void *addr, size_t length, int prot, int flags,
   int fd, off_t offset
);
```

Another example: The **strlen** function returns the length of a null-terminated string. Note that the binary operator **(- a b)** now accepts pointers.

```
(def (strlen int) ((s ptr byte)) (do
    (var start s)
    (loop (peek s) (set s (+ 1 s)))
    (return (- s start))
))
```

The last example is copying data through pointers.

```
;; copy data byte by byte
(def (memcpy void) ((dst ptr byte) (src ptr byte) (n int)) (do
    (loop n (do
        (poke dst (peek src))
        (set dst (+ 1 dst))
        (set src (+ 1 src))
        (set n (- n 1))
    ))
))
```

14.3 The Implementation

14.3.1 Peek and Poke

Data types in our language are either 8-bit or 64-bit. So we need 2 separate bytecode instructions to load data from a pointer, they are **peek** and **peek8** respectively.

```
class CodeGen:
    def peek(self, var, dst):
        self.load_rax(var)
        # mov rax, [rax]
        self.asm_load(CodeGen.A, CodeGen.A, 0)
        self.store_rax(dst)

    def peek8(self, var, dst):
        self.load_rax(var)
        # movzx eax, byte ptr [rax]
        self.buf.extend(b"\x0f\xb6\x00")
        self.store_rax(dst)

    def poke(self, ptr, val):
        ...

    def poke8(self, ptr, val):
        ...
```

The **poke** and **poke8** are used to write to pointers. The assembly for them should be easy for you at this point.

The **byte** data type still occupies 8 bytes in memory, and we always keep the upper bits cleared in 64-bit registers, so we don't have to differentiate 8-bit variable loads and stores from 64-bit.

14.3.2 Referencing Variables

Getting the address of a variable is different depending on whether it's local or non-local. So we'll add 2 bytecode instructions.

```
def pl_comp_ref(fenv: Func, node):
    _, name = node

    flevel, var_tp, var = fenv.get_var(name)
    dst = fenv.tmp()
    if flevel == fenv.level:
        fenv.code.append(('ref_var', var, dst))         # local
    else:
        fenv.code.append(('ref_env', flevel, var, dst)) # non-local
    return ('ptr', *var_tp), dst
```

For a local variable, getting its address is a simple **`lea`** instruction. For non-local variables, we need to get the address of the target frame first.

```
class CodeGen:
    def ref_var(self, var, dst):
        # lea rax, [rbx + var*8]
        ...

    def ref_env(self, level_var, var, dst):
        # mov rax, [rsp + level_var*8]
        self.load_env_addr(level_var)
        # add rax, var*8
        self.buf.extend(b"\x48\x05")
        self.i32(var * 8)
        self.store_rax(dst)
```

Remember that we used a separate stack for local variables from the call stack, this design has a security advantage: you cannot accidentally overwrite the return address with a stray pointer, this defeats a class of security exploits called "return-oriented programming".

14.3.3 Typecastings

Most valid typecastings are no-op since the CPU doesn't care about types.

```
def pl_comp_cast(fenv: Func, node):
    _, tp, value = node
    tp = validate_type(tp)
```

```
    val_tp, var = pl_comp_expr_tmp(fenv, value)

    # to, from
    free = [
        ('int', 'ptr'),
        ('ptr', 'int'),
        ('ptr', 'ptr'),
        ('int', 'byte'),
        ('int', 'int'),
        ('byte', 'byte'),
    ]
    if (tp[0], val_tp[0]) in free:
        return tp, var
    if (tp[0], val_tp[0]) == ('byte', 'int'):
        fenv.code.append(('cast8', var))
        return tp, var

    raise ValueError('bad cast')
```

However, when casting a 64-bit integer to 8-bit, the upper bits should be cleared. This can be accomplished with **`and`** or **`movzx`**.

```
class CodeGen:
    def cast8(self, var):
        # and qword ptr [rbx + var*8], 0xff
        self.asm_disp(b"\x48\x81", 4, CodeGen.B, var * 8)
        self.i32(0xff)
```

14.3.4 Pointer Arithmetic

The **`+`** and **`-`** operators are augmented to handle pointers, just like in C.

Adding an offset to a pointer results in **`ptr + scale * offset`**, where the **`scale`** is the element size, which is either 1 or 8 bytes in our language. This can easily be mapped to the **`lea`** x64 instruction. We'll also add a bytecode instruction for this.

```
def pl_comp_binop(fenv: Func, node):
    op, lhs, rhs = node

    # compile subexpressions
    save = fenv.stack
    t1, a1 = pl_comp_expr_tmp(fenv, lhs)
    t2, a2 = pl_comp_expr_tmp(fenv, rhs)
    fenv.stack = save   # discard temporaries

    # pointers
    if op == '+' and (t1[0], t2[0]) == ('int', 'ptr'):
        # rewrite `offset + ptr` into `ptr + offset`
        t1, a1, t2, a2 = t2, a2, t1, a1
    if op in '+-' and (t1[0], t2[0]) == ('ptr', 'int'):
        # ptr + offset
        scale = 8
        if t1 == ('ptr', 'byte'):
            scale = 1
        if op == '-':
            scale = -scale
        # output to a new temporary
        dst = fenv.tmp()
        fenv.code.append(('lea', a1, a2, scale, dst))
        return t1, dst
    ...
```

When subtracting 2 pointers, the result is divided by the element size. We may need to use the right shift instruction. This is left as an exercise for the reader.

14.4 More Demos

To illustrate the generality of our language, let's examine a more sophisticated demo program.

14.4.1 The Bump Allocator

The first thing is dynamic memory allocation for small-sized data. The **`mmap`** can only allocate memory that is a multiple of the page size, which can be wasteful for data smaller

than 1 page. Thus memory allocators are usually implemented on top of large chunks of memory. Below is a so-called "bump allocator", which allocates memory linearly from a large chunk and never reclaims free memory.

```
(var heap (ptr byte))

; a fake malloc
(def (malloc ptr byte) ((n int)) (do
    (if (not heap) (do
        ; create the heap via mmap()
        (var heapsz 1048576)    ; 1M
        (var prot 3)            ; PROT_READ|PROT_WRITE
        (var flags 0x22)        ; MAP_PRIVATE|MAP_ANONYMOUS
        (var fd -1)
        (var offset 0)
        (var r (syscall 9 0 heapsz prot flags fd offset))
        (set heap (cast (ptr byte) r))
    ))
    ; just move the heap pointer forward
    (var r heap)
    (set heap (+ n heap))
    (return r)
))

; never free anything
(def (free void) ((p ptr byte)) (do))
```

This type of allocator is only useful for some specialized use cases. General-purpose memory allocators are much more complicated.

14.4.2 A String Library

With a memory allocator, we can do more interesting things, such as a string library.

The string library we'll implement doesn't use null-terminated strings like C, instead, we'll add a 16-byte header in front of the string data to store the length and capacity of the string buffer.

```
; allocate a new string.
; the length and capacity are stored before the string data.
; | len | cap | data
; |  8  |  8  | ....
(def (strnew ptr byte) ((cap int)) (do
    (var addr (call malloc (+ 16 cap)))
    (var iaddr (cast (ptr int) addr))
    (poke iaddr 0)
    (poke (+ 1 iaddr) cap)
    (return (+ 16 addr))
))

; free the string
(def (strdel void) ((s ptr byte)) (do
    (call free (- s 16))
))
```

Accessing the length or capacity is simply a pointer dereference.

```
; access the len and the cap
(def (strlen int) ((s ptr byte)) (do
    (var iaddr (cast (ptr int) s))
    (return (peek (- iaddr 2)))
))
(def (strcap int) ((s ptr byte)) (do
    (var iaddr (cast (ptr int) s))
    (return (peek (- iaddr 1)))
))
```

Appending a byte to a string:

```
; append a character to a string, growing it if necessary.
(def (append ptr byte) ((s ptr byte) (ch byte)) (do
    (var len (call strlen s))
    (var cap (call strcap s))
    (if (eq len cap) (do
        ; create a new string with double the capacity
        ; omitted...
    ))
```

```
    ; write the character
    (poke (+ len s) ch)
    ; update the length field
    (poke (cast (ptr int) (- s 16)) (+ 1 len))
    (return s)
))
```

14.4.3 Number to String Conversion

Converting an integer to a string:

```
; reverse a string in place
(def (strrev ptr byte) ((s ptr byte)) (do
    (var l s)
    (var r (- (+ s (call strlen s)) 1))
    (loop (lt l r) (do
        (var t (peek l))
        (poke l (peek r))
        (poke r t)
        (set l (+ l 1))
        (set r (- r 1))
    ))
    (return s)
))

; convert an int to string
; FIXME: negative numbers
(def (str ptr byte) ((i int)) (do
    (var s (call strnew 24))
    (if (eq 0 i) (call append s '0'))
    (loop i (do
        (var d (+ 48 (% i 10)))
        (set i (/ i 10))
        (set s (call append s (cast (byte) d)))
    ))
    (call strrev s)
    (return s)
))
```

14.5 Closing Remarks

If you have followed this book to this point, you'll see that there is no magic from source code to machine code. It's often said that learning C makes you understand the machine, and this is true for assembly and compilers.

There is more about compilers, such as parsing complex grammar, optimization techniques, etc. You should now have the confidence to take on a real compiler book if you want to learn more. Or you may want to add more language features, target alternative CPU architectures, and so on.

This book is part of the "*Build Your Own X*" book series. The goal of this series is to teach basic things using the "*from scratch*" approach. You can find more books on the official website, such as "Build Your Own Redis" and "Build Your Own Database".

https://build-your-own.org