# For those who want to know about low layers. Cコンパイラ作成入門

Rui Ueyama <ruiu@cs.stanford.edu>

2020-03-16

# Introduction.

*This online book is a work in progress. It is not a finished version. Feedback Form*

This book is packed with contents that are rather greedy to include in one book. In this book, we will create a C compiler, which is a program that converts source code written in C into assembly language. The compiler itself will also be developed using C. The immediate goal is to be able to compile your own source code with a self-hosted compiler.

In this book, I have decided to explain various topics in a gradual manner throughout the book so that the difficulty of explaining the compiler does not increase too quickly. The reasons for this are as follows

Compilers can be conceptually divided into multiple stages: parsing, intermediate paths, and code generation. A common textbook approach would be to provide chapters on each topic, but books with such an approach tend to get too narrow and deep in the middle of the story, making it difficult for the reader to follow.

Another disadvantage of the stage-by-stage development method is that you can't run the compiler until all the stages are completed, so you can't notice if your understanding or code is definitely wrong until the whole thing starts running. In the first place, you don't really know what is expected as input for the next stage until you create it yourself, so you don't really know what to output in the stage before that. There is also the problem of not being able to compile any code at all until it is completed, making it difficult to maintain motivation.

In this book, we have decided to take a different approach to avoid that trap. At the beginning of the book, the reader will implement a "proprietary language" with a very simple language specification. The language is so simple that it is not necessary to know much about how to make a compiler when implementing it. After that, the reader will work through the book to add features to the "original language" and eventually develop it into something that is consistent with C.

In such an incremental development method, the compiler is built step by step, with small commits. In this development method, the compiler is always "finished" in a sense at every commit. At one stage it may be just a calculator, at another it may be a very limited subset of C, and at yet another it may be almost a C language. The point is that at every stage, the goal is to have a language with a reasonable specification that matches the level of completion at that point. We do not try to make some functions more C-like during development.

Data structures, algorithms, and computer science knowledge will also be explained sequentially according to the development stage.

Incremental development achieves the goal that at any point in the reading of this book, the reader has a comprehensive knowledge of how to build a reasonable language at that level. This is much better than being extremely knowledgeable about only a few topics in compiler creation. This is much better than being extremely knowledgeable about only a few topics in compiler creation, and by the end of the book, you will have a thorough knowledge of all topics.

It is also a book that explains how to write a large program from scratch. The skill of writing a large program is a unique skill that is different from learning data structures and algorithms, but I don't think there are many books that explain such things. There are not many books that explain such skills, and even if they do, it is difficult to know whether a development method is good or bad unless you actually experience it. This book is designed in such a way that the process of developing your own language into C becomes an actual experience of a good development method.

If the author is successful, this book will teach the reader not only compiler writing techniques and knowledge of the CPU instruction set, but also how to break a large program into small steps, software testing techniques, version control techniques, and even how to prepare for an ambitious project like compiler writing. You will also learn how to build a large program in small steps, software testing techniques, version control techniques, and even how to prepare yourself to tackle an ambitious project like compiler building.

The intended audience for this book is the average C programmer, not necessarily a super C programmer who is familiar with the C language specification. You don't need to be a super C programmer who knows the C language specification very well, but just enough to understand pointers and arrays, and be able to read small C programs written by others, at least if you take the time to do so.

In writing this book, I have tried to explain as much as possible about the language and CPU specifications, not only explaining the specifications, but also explaining why such designs were chosen. I also tried to make the book enjoyable to read by interspersing columns about compilers, CPUs, the computer industry, and its history that would attract the reader's interest.

Creating a compiler is a very enjoyable task. As you continue to develop your own language, which in the beginning was only ridiculously simple, it will quickly grow to become so C-like that you will be amazed at how well it works, as if by magic. I am often surprised to find that large test codes that I didn't think would compile well at the time compile without errors and work perfectly correctly. Such code is not immediately understandable to me when I look at the assembly of the compilation results. Sometimes, I even feel that my compiler has an intelligence that surpasses me as the author. Compilers are programs that, even if you know how they work, you still feel a sense of wonder as to why they work so well. I am sure you will be fascinated by it.

Now that we've got that out of the way, let's dive right into the world of compiler development with the author!

### Column: Why C?

Of the many programming languages available, why did we choose C for this book? Or why not a homegrown language? In this regard, there is no reason why it absolutely has to be C. However, if I had to choose some language to learn how to create a compiler that outputs native code, I think C is one of the not so many reasonable choices.

In an interpreted language, you can't learn much about the lower layers. On the other hand, C usually compiles into assemblies, so by making a compiler, you can learn about C itself as well as the instruction set of the CPU and how the program works.

Since C is widely used, once you get your compiler working properly, you can compile and play with third-party source code that you download from the net. For example, you may be able to build and play with a mini-Unix xv6. If the compiler is complete enough, it should be possible to compile even the Linux kernel. This kind of fun is not possible with minor languages or home-made languages.

One statically typed language that compiles to a native machine language like C, and is at least as widely used as C, is C++. However, C++ is not a realistic option because the language specification is so huge that it is impossible to make a casual home-made compiler.

Designing and implementing an original language is good in the sense that it hones your sense of language design, but it also has its pitfalls. It is possible to avoid implementing parts that are troublesome by avoiding them in the language specification. This is not the case with languages such as C, where the language specification is given as a standard. I think this restriction is rather good in terms of learning.

## Notation of this book

Functions, expressions, and commands are displayed in monospaced font in the text, such as main, foo=3, and make.

Code that spans multiple lines is displayed in a frame using a monospaced font, as shown below.

```
int main() {
 printf("Hello world!\n");
 return 0;
}
```

In the case of shell commands where the framed code is intended to be typed by the user as is, the line starting with $ represents a prompt. Type anything after $ in that line into the shell (do not type $ itself). The lines other than $ represent the output from the command you entered. For example, the block below is an example of what happens when a user types the string make and presses enter. the output from the make command is make: Nothing to be done for `all'. The output from the make command is make: Nothing to be done for `all'.

```
$ make
make: Nothing to be done for `all'.
```

## The development environment assumed in this document

This book assumes a 64-bit Linux environment running on a so-called normal PC such as Intel or AMD. Please install development tools such as gcc and make beforehand according to the distribution the reader is using; if you are using Ubuntu, you can install the commands used in this book by executing the following commands.

```
$ sudo apt update
$ sudo apt install -y gcc make git binutils libc6-dev
```

While macOS is fairly compatible with Linux at the assembly source level, it is not fully compatible (specifically, it does not support a feature called "static linking"). While it is not impossible to write a macOS-compatible C compiler according to the contents of this book, you will be plagued with various minor incompatibilities when you actually try. It is not recommended to learn C compiler techniques and the differences between macOS and Linux at the same time. It is not recommended to learn C compiler writing techniques and the differences between macOS and Linux at the same time, because if something doesn't work, you won't know which one you understand wrong.

If you have never prepared a Linux virtual environment before, please refer to Appendix3, which describes how to create a development environment using Docker. If you are new to preparing a Linux virtual environment, please refer to Appendix 3 for instructions on creating a development environment using Docker.

Windows is incompatible with Linux at the assembly source level. The Windows Subsystem for Linux (WSL) application is a Linux-compatible environment. Windows Subsystem for Linux (WSL) is the Linux compatible environment. When practicing the contents of this book on Windows, please install WSL and proceed with development in it.

**Column: Cross Compiler**

The machine on which the compiler runs is called the "host", and the machine on which the code output by the compiler runs is called the "target". In this book, both are 64-bit Linux environments, but the host and target do not necessarily have to be the same.

A compiler that has a different host and target is called a cross compiler. For example, the compiler that runs on Windows that generates the Raspberry Pi executable is a cross compiler. A cross compiler is often used when the target machine is too poor or special to run the compiler.

## About the Author

Rui Ueyama (@rui314). Rui Ueyama (@rui314) is the original author and current maintainer of the fast linker lld, which has been adopted as the standard linker for creating executables in many operating systems and projects, including Android (version Q and later), FreeBSD (12 and later), Nintendo Switch, Chrome and Firefox. It has been adopted as the standard linker for creating executables in many operating systems and projects, such as Android (version Q or later), FreeBSD (12 or later), Nintendo Switch, Chrome, Firefox, etc. (Therefore, there is a high possibility that the binary created by the tool written by the author is in the reader's computer. He is also the author of the compact C compiler 8cc. His essays on software are mainly in NOTE.

**Column: Compiler to Compile Compiler**

It is not uncommon to find self-referential situations, such as a C compiler being written in C. Outside of C, many language implementations are written using the language itself.

If you already have an implementation of language X, there is no logical contradiction in creating a new X compiler using the language itself. If you want to do self-hosting, you can simply continue development with the existing compiler and switch when you have completed your own. That is exactly what we are trying to do in this book.

But what if you don't have an existing compiler? Then you have no choice but to write it using another language. When you write your first compiler for X with the intention of self-hosting, you need to write it using the existing Y language, which is different from X. When the compiler is complete, you need to rewrite the compiler itself from Y to X.

If we trace the genealogy of the compilers of modern complex programming languages back to another compiler that was used to compile an implementation of that language, we will eventually arrive at a simple assembler that someone wrote directly in machine language at the dawn of computers. I don't know if there was a single assembler or multiple assemblers, the ultimate ancestor of all existing language implementations, but I am sure that today's compilers originate from a very small number of ancestors. Since non-compiler executables are usually compiler-generated files, almost all existing executables are indirect descendants of that primitive assembler. This is an interesting story, like the origin of life.

[redacted]

# Machine Language and Assembler

The goal of this chapter is to give you a rough idea of the components that make up a computer and what kind of code should be output from our C compiler. We will not go into the specific instructions of the CPU yet. It is important to grasp the concepts first.

## CPU and memory

The components that make up a computer can be roughly divided into the CPU and memory. Memory is a device that can hold data, and the CPU is a device that performs some kind of processing while reading and writing to the memory.

Conceptually, to the CPU, memory looks like a huge array of randomly accessible bytes, and when the CPU accesses memory, it specifies the number of bytes it wants to access, which is called the "address. For example, "read 8 bytes of data from address 16" means to read 8 bytes of data from the 16th byte of memory, which looks like an array of bytes. The same thing can also be said as "reading 8 bytes of data from address 16".

Both the program that the CPU executes and the data that the program reads and writes are stored in memory. Then the next instruction is read and executed. The address of the currently executing instruction is called the "program counter" (PC) or "instruction pointer" (IP), and the program format itself that the CPU executes is called "machine code.

The program counter does not necessarily advance linearly to the next instruction only; a type of CPU "branch instruction" can be used to set the program counter to any address other than the next instruction. This is how if statements, loops, etc. are realized. Setting the program counter to a location other than the next instruction is called "jumping" or "branching".

CPUs have a small number of data storage areas in addition to the program counter. For example, Intel and AMD processors have 16 areas that can hold 64-bit integers. These areas are called "registers". Memory is a device external to the CPU, and it takes some time to read and write to it, but registers are internal to the CPU and can be accessed without delay.

Most machine languages are formatted in such a way that some operation is performed using the values of two registers, and the result is written back to the registers. Therefore, the execution of a program proceeds as follows: the CPU reads data from memory into the registers, performs some operation between the registers, and then writes the result back to memory.

The instructions for a particular machine language are collectively referred to as "instruction set architecture" (ISA) or "instruction set. There is no single instruction set, and each CPU can design its own instruction set as it likes. However, there are not many variations of instruction sets, because the same program cannot run without compatibility at the machine language level. x86-64 is one of the major instruction sets used in PCs, and is called x86-64 by Intel and its compatible chip manufacturer AMD. Although x86-64 is one of the major instruction sets, it is not the only one that dominates the market. For example, the iPhone and Android use the ARM instruction set.

### Column: Names of the x86-64 instruction set

x86-64 is also sometimes referred to as AMD64, Intel 64, x64, etc. There is a historical reason why the same instruction set has multiple names like this.

The x86 instruction set was created by Intel in 1978, but it was AMD that extended it to 64 bits. in 2000, when 64 bit processors were becoming a necessity, Intel was working on a whole new instruction set called Itanium, and did not dare to work on a 64 bit version of x86 to compete with it. Intel was working on a completely new instruction set, Itanium, in 2000 when the need for 64-bit processors became apparent, and was not working on the 64-bit version of x86 that would have competed with it. AMD took advantage of this gap and released a 64-bit version of the x86 specification. That was x86-64. After that, AMD changed the name of x86-64 to AMD64, probably for branding reasons.

After the Itanium failure, Intel was left with no choice but to make a 64-bit version of x86, but by that time there were a good number of actual AMD64 chips available, so it was difficult to create an extended instruction set that was not similar to the AMD64, and Intel decided to use an AMD-compatible instruction set. It is said that there was also pressure from Microsoft to maintain compatibility. The name IA-32e (Intel Architecture 32 extensions), instead of 64, was chosen to indicate that the main focus of 64-bit CPUs would be on Itanium. The name IA-32e (Intel Architecture 32 extensions) instead of 64 seems to indicate that the main focus of the 64-bit CPU is Itanium, and that the unsuccessful instruction set is still being worked on. After that, Intel decided to abandon Itanium altogether, and IA-32e was renamed to Intel 64, which is the usual name, although Microsoft does not like too long a name, and calls x86-64 x64.

For the above reasons, x86-64 has a lot of different names.

In open source projects, the name x86-64 is often preferred because it does not contain the name of a specific company. In this book, the name x86-64 is used consistently.

## What is an assembler?

Since machine language is read directly by the CPU, only the convenience of the CPU is taken into consideration, and not the ease of use for humans. Writing such machine language with a binary editor is not impossible, but it is very hard work. This is where the assembler was invented. Assembly is a language that corresponds almost directly to machine language on a one-to-one basis, but it is much easier for humans to read than machine language.

For compilers that output native binaries rather than virtual machines or interpreters, the goal is usually to output assembly. Even compilers that appear to output machine language directly, in a common configuration, start an assembler in the background after outputting assembly. The C compiler we will build in this book also outputs assembly.

Converting assembly code into machine language is sometimes called "compiling", but sometimes it is specially called "assembling", emphasizing that the input is assembly.

If you have not seen assembly before, now is a good time to take a look. If you have never seen an assembly before, now is a good time to take a look: use the `objdump` command to disassemble a suitable executable file and display the machine language contained in the file as an assembly. The following is the result of disassembling the `ls` command.

```
$ objdump -d -M intel /bin/ls
/bin/ls:      file format elf64-x86-64

Disassembly of section . init:

0000000000003d58 <_init@@Base>:
 3d58: 48 83 ec 08             sub     rsp,0x8
 3d5c: 48 8b 05 7d b9 21 00 mov rax,QWORD PTR [rip+0x21b97d].
 3d63: 48 85 c0                test    rax,rax
 3d66: 74 02                   je      <_init@@Base+0x12>
 3d68: ff d0                   call    rax
 3d6a: 48 83 c4 08             add     rsp,0x8
 3d6e: c3                      ret
...
```

In the author's environment, the `ls` command contains about 20,000 machine language instructions, so the disassembled result is also nearly 20,000 lines long. Only the first part is shown here.

In an assembly, the basic structure is one line per machine word. As an example, let's look at the following line.

```
 3d58: 48 83 ec 08             sub    rsp,0x8
```

What is the meaning of this line? The 3d58 is the address of the memory that contains the machine language. This means that when the ls command is executed, the instruction in this line is placed at 0x3d58 in memory, and this instruction will be executed when the program counter is 0x3d58. The next four hexadecimal numbers are the actual machine language, which the CPU reads and executes as an instruction. sub rsp,0x8 is the assembly corresponding to the machine language instruction. The CPU instruction set is described in a separate chapter, but this instruction subtracts 8 from the register named RSP The CPU instruction set is explained in a separate chapter, but this instruction subtracts (subtracts) 8 from the register RSP.

C and its corresponding assembler

## A simple example

To get an idea of what kind of output the C compiler is generating, let's compare the C code with the corresponding assembly code. As the simplest example, let's consider the following C program.

```
int main() {
 return 42;
}
```

Assuming that the file in which this program is written is test1.c, we can compile it as follows to see that main actually returns 42.

```
$ cc -o test1 test1.c
$ . /test1
$ echo $?
42
```

In C, the value returned by the main function is the exit code of the program as a whole. The exit code of the program is not displayed on the screen, but it is implicitly set in the shell's $? variable in the shell, so you can print $? immediately after the end of the command, you can see the exit code of the command. Here we can see that 42 is correctly returned.

Now, the corresponding assembly program for this C program is as follows

```
. . intel_syntax noprefix
. . globl main
main:
         mov rax, 42
         ret
```

In this assembly, the global label main is defined, and the label is followed by the code for the main function. Here, the value of 42 is set in a register called RAX and returned from main. There are a total of 16 registers that can hold integers, including RAX, but the value in RAX is promised to be the return value of the function when it returns from the function, so the value is set in RAX here.

Let’s try to actually assemble and run this assembly program. The extension of the assembly file is .s, so write the above assembly code in test2.s and execute the following command.

```
$ cc -o test2 test2.s
$ . /test2
$ echo $?
42
```

Just like in C, 42 is now the exit code.

Roughly speaking, a C compiler is a program that outputs an assembly such as test2.s when it reads C code such as test1.c.

## Example involving a function call

As a more complex example, let’s see how code with function calls can be converted to assembly.

Function calls are different from mere jumps in that after the called function finishes, it must return to the place where it was originally executed. The address where the function was originally executed is called the ”return address”. If there is only one level of function call, the return address can be stored in an appropriate register on the CPU, but since function calls can be as deep as you want, the return address needs to be stored in memory. In practice, the return address is stored on the stack in memory.

A stack can be implemented using only one variable, which holds the address at the top of the stack. In order to support programming with functions, x86-64 supports a register dedicated to the stack pointer and instructions that use the register. Stacking data on the stack is called "pushing", and removing data from the stack is called "popping".

Now, let's look at a real-world example of a function call. Consider the following C code.

```
int plus(int x, int y) {
 return x + y;
}

int main() {
 return plus(3, 4);
}
```

The corresponding assembly for this C code will look like this

```
. . intel_syntax noprefix
. . globl plus, main

plus:
         add rsi, rdi
         mov rax, rsi
         ret

main:
         mov rdi, 3
         mov rsi, 4
         call plus
         ret
```

The first line is an instruction that specifies the syntax of the assembly, and the second line, starting with `.globl`, tells the assembly that the two functions, `plus` and `main`, are not file-scoped but are visible to the whole program. You can ignore this for now.

First, let's look at `main`, from which C calls `plus with` arguments. In the assembler, the first argument is promised to be in the RDI register and the second argument in the RSI register, so the first two lines of `main` set the values accordingly.

A `call is an` instruction that calls a function. Specifically, `call does the` following

- Pushes the address of the next instruction of `call` (`ret` in this case) onto the stack
- Jump to the address given as the argument of `call`

Therefore, when the call instruction is executed, the CPU will start executing the `PLUS` function.

Look at the `plus` function, which has three instructions.

`add` is an instruction that performs addition. In this case, the result of adding the RSI and RDI registers will be written to the RSI register. x86-64 integer arithmetic instructions usually accept only two registers, so the result will be stored in the form of overwriting the value of the first argument register.

The return value from the function was to be put into RAX. Therefore, since we want to keep the result of the addition in RAX, we need to copy the value from RSI to RAX. Here, we use the `mov` instruction to do so. `mov` is an abbreviation of move, but it does not actually move the data, but simply copies it.

At the end of the `plus` function, we call `ret to` return from the function. Specifically, `ret does the` following

- Pop one address from the stack
- Jump to that address

In other words, `ret` is an instruction that undoes what call has done and resumes the execution of the calling function. Thus, `call` and `ret are` defined as a pair of instructions.

The return from `plus is the ret` instruction of `main. In the` original C code, the return value of `plus was` returned directly from `main`. Here, the return value of `plus is in` RAX, so by returning from `main` as it is, it can be used as the return value from `main`.

## Summary of this chapter

This chapter has given an overview of how a computer works inside and what a C compiler is supposed to do. When we look at assemblies and machine language, it looks like a jumbled mass of data that is far removed from C. However, many readers may think that it actually reflects the structure of C in a surprisingly straightforward way.

Since we have not yet explained most of the specific machine language in this book, you may not be able to understand the meaning of individual instructions in the assembly code displayed by `objdump`, `but` you can imagine that each instruction does not do much. It is enough to get such a feeling at this stage of the chapter.

The main points of this chapter are summarized below in bullet points.

- The CPU proceeds with program execution by reading and writing memory.
- Both the program that the CPU executes and the data that the program handles are stored in memory, and the CPU reads machine language instructions from memory in order and executes them.
- The CPU has a small storage area called a register, and many machine words are defined as operations between registers.
- Assembly is a human-readable version of machine language, and C compilers usually output assembly.
- A function in C is also a function in assembly
- Function calls are implemented using the stack.

**Column:Online Compiler**

Looking at the C code and its compilation results is a good way to learn assembly language, but it is surprisingly tedious to edit the source code, compile it, and check the output assembly many times. There is a very good website that can save you that trouble. If you want to see what kind of assembly your C code will be converted into, this is a good place to start. If you want to see what kind of assembly your C code will be converted into, this site is a good place to start.

[redacted]

# Creating a calculator-level language

In this chapter, as the first step in creating a C compiler, we will support the four arithmetic operations and other arithmetic operators so that we can compile expressions such as the following

```
30 + (4 - 2) * -5
```

This may seem like a trivial goal, but it is actually quite a difficult one. Mathematical expressions have a structure in which expressions in parentheses take precedence, or multiplication takes precedence over addition, and unless you understand this structure in some way, you will not be able to perform calculations correctly. However, a formula given as input is just a flat string of characters, not structured data. In order to evaluate the formula correctly, we need to analyze the sequence of characters and successfully derive the structure hidden in it.

These parsing problems are quite difficult if you try to solve them without any prerequisite knowledge. In fact, these problems were considered difficult in the past, and there was a lot of research and development of various algorithms, especially from the 1950s to the 1970s. Thanks to these efforts, parsing is no longer a very difficult problem if you know how to do it.

This chapter describes one of the most common algorithms for parsing, recursive descent parsing, which is also used by many C/C++ compilers such as GCC and Clang.

In addition to compilers, the need to read text that has some structure to it often arises in programming. The techniques you will learn in this chapter can be used for such problems as well. The parsing techniques that you will learn in this chapter are not exaggerated, but rather, they are a way of life. Read this chapter, understand the algorithms, and put the parsing techniques in your toolbox as a programmer.

## Step 1: Create a language to compile a single integer.

Consider a subset of the simplest C language. What kind of language do you imagine? A language with only a `main` function? Or a language with only a single expression? When you get right down to it, I think that a language with only one integer is the simplest subset you can think of.

In this step, we will first implement its simplest language.

The program we will create in this step is a compiler that reads a single number from the input and outputs an assembly that ends with that number as the exit code for the program. In other words, the input is simply a string like 42, and we will create a compiler that reads it and outputs an assembly like the following

```
. . intel_syntax noprefix
. . globl main

main:
        mov rax, 42
        ret
```

. . intel_syntax noprefix″ is an assembler command to select the Intel notation used in this book among several assembly writing methods. Please make sure to include this line at the beginning of your compiler. The rest of the lines are as explained in the previous chapter.

The reader may think, ″This program cannot be called a compiler. To be honest, I think so too. However, this program accepts a language consisting of a single numeric value as input and outputs the code corresponding to that value, which by definition is a fine compiler. Such a simple program can quickly become quite difficult to modify, so let's complete this step first.

In fact, this step is very important in terms of the overall development procedure. This step is actually very important from the point of view of the entire development process, because we will be using what we create in this step as a skeleton for future development. In this step, in addition to creating the compiler itself, we will also create the build file (Makefile), automated tests, and set up the git repository. Let's take a look at these tasks one by one.

The C compiler in this book is called 9cc, where cc is an abbreviation for C compiler. 9 does not have any particular meaning, but since my previous C compiler was called 8cc, I decided to name it 9cc as a follow-up to that. Of course, you can give it any name you like. The name can be changed later, including in the GitHub repository, so there is no problem in starting with an appropriate name.

### Column: Intel Notation and AT&T Notation

In addition to the Intel notation used in this book, AT&T notation is also widely used in assembler notation, especially on Unix. gcc and objdump output assemblies in AT&T notation by default.

In AT&T notation, the result register comes as the second argument. Therefore, in a two-argument instruction, the arguments are written in reverse order. The register name is written as `%rax with a` %prefix. The numeric value is written as `$42 with a $` prefix.

Also, when referring to memory, the expression is written in a unique notation, using `()` instead of `[]`. Some examples are given below for contrast.

```
mov rbp, rsp    // Intel
mov %rsp, %rbp // AT&T

mov rax, 8      // Intel
mov $8, %rax    // AT&T

mov [rbp + rcx * 4 - 8], rax // Intel
mov %rax, -8(rbp, rcx, 4)     // AT&T
```

The Intel notation is used in the Intel instruction set manual, so it has the advantage that the description in the manual can be written directly into the code. The expressive power of both AT&T and Intel notation is the same. The power of expression is the same for both AT&T and Intel notation, and the machine language instruction strings generated are identical regardless of which notation is used.

## Creating the compiler body

We usually give the compiler the input as a file, but in this case we will give the code directly as the first argument of the command, since it is not convenient to open and read the file. A C program that reads the first argument as a number and embeds it in a canned assembly can be easily written as follows

```
#include < stdio.h>.
#include < stdlib.h>.

int main(int argc, char **argv) {
 if (argc ! = 2) {
 fprintf(stderr, "The number of arguments is not correct \n");
 return 1;
 }

 printf(". intel_syntax noprefix\n");
 printf(". globl main\n");
 printf("main:\n");
 printf(" mov rax, %d\n", atoi(argv[1]));
 printf(" ret\n");
 return 0;
}
```

Create an empty directory called `9cc` and create a file called `9cc.c` in it with the above contents. After that, run 9cc as shown below to see how it works.

```
$ cc -o 9cc 9cc.c
$ . /9cc 123 > tmp.s
```

In the first line, we compile `9cc.c` and create an executable called `9cc`.

```
$ cat tmp.s
. . intel_syntax noprefix
. . globl main
main:
 mov rax, 123
 ret
```

As you can see, it is generated well. Passing the resulting assembly file to the assembler, we can create the executable.

In Unix, `cc` (or `gcc`) is supposed to be a front-end for many languages, not just C and C++, and it judges the language by the extension of the given file and starts the compiler or assembler. So here, we can pass an assembler file with `.s` extension to `cc` to assemble, just as we did when we compiled 9cc. The following is an example of assembling and running the generated executable.

```
$ cc -o tmp tmp.s
$ . /tmp
$ echo $?
123
```

In the shell, the exit code of the previous command is accessible in the variable `$?` variable. In the above example, the number 123 is displayed, which is the same as the argument given to 9cc. Try giving 9cc a number other than 123 in the range of 0-255 (the Unix process exit code is supposed to be 0-255) and see if 9cc actually works.

## Creating automated tests

Many readers may have never written tests in their hobby programming, but in this book, every time you extend the compiler, you will write code to test the new code. Writing tests may seem tedious at first, but you will soon learn to appreciate them. If you didn't write the test code, you would eventually have to run the same kind of tests by hand every time to check the behavior, but it is much more tedious to do it by hand.

I think a large part of the impression that writing tests is tedious comes from the fact that test frameworks are exaggerated and the philosophy of testing is sometimes dogmatic. Testing frameworks such as JUnit, for example, have many useful features, but it takes a lot of time to install and learn how to use them. Therefore, we will not introduce such test frameworks in this chapter. Instead, we will write a very simple hand-written "test framework" in a shell script and use it to write tests.

The following is the shell script test.sh for testing. The shell function `assert` takes two arguments, the value of the input and the value of the expected output, and actually assembles the result of 9cc and compares the actual result with the expected value. In the shell script, after defining the `assert` function, we use it to make sure that both 0 and 42 compile correctly.

```
#! /bin/bash
assert() {
 expected="$1"
 input="$2"

 . /9cc "$input" > tmp.s
 cc -o tmp tmp.s
 . /tmp
 actual="$? "

 if [ "$actual" = "$expected" ]; then
     echo "$input => $actual"
 else
     echo "$input => $expected expected, but got $actual"
     exit 1
 fi
}

assert 0 0
assert 42 42

echo OK
```

Create `test.sh` with the above contents and make it executable by running `chmod a+x test.sh. Let's` actually run `test.sh`. If no errors occur, `test.sh will` exit with `OK` at the end, as shown below.

```
$ . /test.sh
```

```
0 => 0
42 => 42
OK.
```

If an error occurs, `test.sh` will not print `OK`. Instead, `test.sh will` print the expected and actual values for the failed test as follows

```
$ . /test.sh
0 => 0
42 expected, but got 123
```

When you want to debug a test script, give bash the option `-x` and run the script. With the `-x` option, bash will show a trace of the execution as follows

```
$ bash -x test.sh
+ assert 0 0
+ expected=0
+ input=0
+ cc -o 9cc 9cc.c
+ . /9cc 0
+ cc -o tmp tmp.s
+ . /tmp
+ actual=0
+ '[' 0 '! ='' 0 ']''
+ assert 42 42
+ expected=42
+ input=42
+ cc -o 9cc 9cc.c
+ . /9cc 42
+ cc -o tmp tmp.s
+ . /tmp
+ actual=42
+ '[' 42 '! ='' 42 ']''
+ echo OK
OK.
```

The "test framework" we will use throughout this book is simply a shell script like the one above. This script may seem too simple compared to a full-fledged testing framework such as JUnit, but the simplicity of this shell script is balanced by the simplicity of 9cc itself, so it is preferable to have it this simple. The point of automated testing is just to be able to run the code you have written in one shot and compare the results mechanically, so don't think too hard about it.

## Building with make

Throughout this book, you will be building 9cc hundreds or even thousands of times, and since the task of creating the 9cc executable and then running the test scripts is the same every time, it is convenient to let the tools do the work for you. The make command is the standard one for this purpose.

When executed, make reads a file named Makefile in the current directory and executes the commands written in it. makefile consists of a rule ending with a colon and a sequence of commands for that rule. The following Makefile is to automate the commands we want to execute in this step.

```
CFLAGS=-std=-c11 -g -static

9cc: 9cc.c

test: 9cc
         . /test.sh

Clean:
         rm -f 9cc *.o *~ tmp*

.PHONY: test clean
```

Create the above file in the same directory where you have 9cc.c, with the file name Makefile. Since make understands file dependencies, it is not necessary to run make after modifying 9cc.c and before running make test. Only if the 9cc executable is older than 9cc.c, make will build 9cc before running the test.

Make clean is a rule to delete temporary files. Temporary files can be rm'd by hand, but it can be troublesome if you accidentally delete files you don't want to delete, so we write these utilities in the Makefile.

Note that the indentation in the Makefile must be a tab character. Four or eight spaces will result in an error. This is just bad syntax, but make is an old tool developed in the 1970s, and this is traditionally the way it is.

Be sure to pass the -static option to cc. This option will be explained in the chapter called Dynamic Linking. There is no need to think about the meaning of this option for now.

## Version control with git

In this book, we will use git as our version control system. In this book, we will build the compiler step by step, and for each step, you should create a commit in git and write a commit message. If you want to write more than one line of detailed explanation, leave a blank line after the first line and write the explanation after that.

The only files you need to version control in git are the ones you generate by hand; you don't need to include files that are generated as a result of running 9cc because you can generate them again by running the same command. Rather, you need to remove these files from version control and keep them out of the repository, because if you include them, the changes per commit will be unnecessarily long.

In git, you can write a `.gitignore` file that describes the pattern of files to be removed from version control. In the same directory where you have `9cc.c`, create a `.gitignore` file with the following contents and set git to ignore temporary files, editor backup files, etc. Set up git to ignore temporary files, editor backup files, etc.

```
*~
*.o
tmp*
a.out
9cc
```

If you are new to git, you should tell git your name and email address. The name and email address you give git will be recorded in the commit log. Below is an example of setting the author's name and email address. Readers are encouraged to set their own name and email address.

```
$ git config --global user.name "Rui Ueyama"
$ git config --global user.email "ruiu@cs.stanford.edu"
```

In order to create a commit in git, you first need to add the files that have changed using `git add`. Since this is your first commit, you'll first create a git repository with `git init`, and then add all the files you've created so far with `git add`.

```
$ git init
Initialized empty Git repository in /home/ruiu/9cc
$ git add 9cc.c test.sh Makefile . . gitignore
```

Then commit it with `git commit`.

```
$ git commit -m "Create a compiler that compiles a single integer"
```

Use the `-m` option to specify the commit message. If there is no `-m` option, `git will` launch the editor. You can check that the commit was successful by running `git log -p as` follows

```
$ git log -p
commit 0942e68a98a048503eadfee46add3b8b9c7ae8b1 (HEAD -> master)
Author: Rui Ueyama <ruiu@cs.stanford.edu>
Date:    Sat Aug 4 23:12:31 2018 +0000

 Create a compiler to compile a single integer

diff --git a/9cc.c b/9cc.c
new file mode 100644
index 0000000... e6e4599
--- /dev/null
+++ b/9cc.c
@@ -0,0 +1,16 @@
+#include < stdio.h>.
+#include < stdlib.h
+
+int main(int argc, char **argv) {
+ if (argc ! = 2) {
...
```

Finally, upload the git repository you have created to GitHub. There is no particular reason to upload to GitHub, but there is no reason not to, and GitHub is useful as a backup for your code. To upload to GitHub, create a new repository (in this example, we created a repository called `9cc` with a user named rui314) and add it as a remote repository with the following command

```
$ git remote add origin git@github.com:rui314/9cc.git
```

You can then run `git push to` push the contents of your repository to GitHub, and after running `git push`, open GitHub in your browser and check that your source code has been uploaded.

This completes the first step of creating a compiler. The compiler in this step is a program that is too simple to be called a compiler, but it is a fine program that contains all the elements necessary for a compiler. From now on, we will continue to extend this compiler, and believe it or not, we will make it into a great C compiler. First of all, please savor the fact that the first step has been completed.

### Reference implementation

- f722daaaae060611

## Step 2: Create a compiler that can add and subtract.

In this step, we will extend the compiler created in the previous step to accept not only values such as `42`, but also expressions that include addition and subtraction such as `2+11` and `5+20-4`.

An expression like `5+20-4` can be calculated when compiling and the resulting number (`21` in this case) can be embedded in the assembly, but that would make it look like an interpreter instead of a compiler, so we need to output an assembly that does the addition and subtraction at runtime. The assembly instructions that perform addition and subtraction are `add` and `sub`. `add` takes two registers, adds their contents, and writes the result to the first argument register. `sub` is the same as `add`, but performs subtraction. Using these instructions, `5+20-4` can be compiled as follows

```
. . intel_syntax noprefix
. . globl main

main:
         mov rax, 5
         add rax, 20
         sub rax, 4
         ret
```

In the above assembly, `mov` sets RAX to 5, then adds 20 to RAX, and subtracts 4. The value of RAX at the time `ret` is executed should be `5+20-4`, or 21. Let's run it and check it out. Save the above file to `tmp.s`, assemble it, and run it.

```
$ cc -o tmp tmp.s
$ . /tmp
$ echo $?
21
```

The 21 is now displayed correctly as shown above.

Now, how can we create this assembly file? If we think of this expression with addition and subtraction as a "language", this language can be defined as follows

- There's one number at the beginning.
- It is followed by zero or more "terms".
- A term is either a `+` followed by a number or a `-` followed by a number.

If we reduce this definition to a straightforward C code, we get a program like the following.

```
#include < stdio.h>.
#include < stdlib.h>.

int main(int argc, char **argv) {
 if (argc ! = 2) {
 fprintf(stderr, "The number of arguments is not correct \n");
 return 1;
 }

 char *p = argv[1];

 printf(". intel_syntax noprefix\n");
 printf(". globl main\n");
 printf("main:\n");
 printf(" mov rax, %ld\n", strtol(p, &p, 10));

 while (*p) {
 if (*p == '+') {
       p++;
 printf(" add rax, %ld\n", strtol(p, &p, 10));
 Continue;
     }

 if (*p == '-') {
       p++;
 printf(" sub rax, %ld\n", strtol(p, &p, 10));
 Continue;
     }

 fprintf(stderr, "Unexpected character: '%c'\n", *p);
 return 1;
 }

 printf(" ret\n");
 return 0;
}
```

The first half of the program and the `ret` line are the same as before, although it is a bit longer. The first half of the program and the ret line are the same as before, with the addition of the code for reading the terms in the middle. Since this is not a program to read a single number, we need to know how far we have read after reading the number. `atoi` does not return the number of characters read, so we will not know where to start reading the next term in `atoi`. Therefore, here I used the `strtol` function from the C standard library.

After `strtol` reads a numeric value, it updates the pointer of the second argument to point to the next character after the last character read. Thus, after reading one number, if the next character is `+` or `-`, `then p` should point to that character. The above program uses this fact to read terms one after the other in the `while` loop, and output one line of assembly each time a term is read.

Now, let's run this modified version of the compiler, and after updating the `9cc.c` file, we can create a new 9cc file just by running `make`. An example run is shown below.

```
$ make
$ . /9cc '5+20-4'
. . intel_syntax noprefix
. . globl main
main:
 mov rax, 5
 add rax, 20
 sub rax, 4
 ret
```

It looks like the assembly is being output successfully. To test this new feature, let's add one more line of `test` to `test.sh` as follows.

```
assert 21 "5+20-4"
```

Once you have done this, it is time to commit your changes to git. To do so, run the following command.

```
$ git add test.sh 9cc.c
$ git commit
```

Run `git commit to` launch the editor, write "Add addition and subtraction," save it, and exit the editor; use the `git log -p` command to verify that your commit did what you expected it to do. Finally, run `git push to` push the commit to GitHub, and you're done with this step!

### Reference implementation

- afc9e8f05faddf05

## Step 3: Install the tokenizer.

The compiler we created in the previous step has one drawback. If the input contains any whitespace characters, it will throw an error at that point. For example, if you give it the string 5 - 3 with spaces, it will find the spaces when it is trying to read the + or - and will fail to compile.

```
$ . /9cc '5 - 3' > tmp.s
Unexpected character: ' '.
```

There are several ways to solve this problem; one obvious way is to skip over the whitespace characters before trying to read the + or -. One obvious way is to skip the whitespace before trying to read the + or -. There is nothing wrong with this way of doing things, but in this step we will use a different method to solve the problem. The way to do this is to break the input into words before reading the expression.

Just as in Japanese and English, math expressions and programming languages can be thought of as being made up of sequences of words. For example, 5+20-4 can be thought of as being made up of five words: 5, +, 20, -, and 4. These "words" are called "tokens" (tokens). The spaces between the tokens are only there to separate the tokens, and are not part of the word. Therefore, it would be natural to remove whitespace when splitting a string into tokens. Splitting a string into a sequence of tokens is called "tokenizing".

Dividing a string into a sequence of tokens has other advantages as well. When you divide an expression into tokens, you can classify the tokens and give them types. For example, + and - are symbols such as + and - as you can see, while the string 123 means the number 123. By interpreting each token when tokenizing, instead of just splitting the input into a string, there is less to think about when consuming the token sequence.

In the current syntax of expressions that can be added and subtracted, there are three types of tokens: +, -, and numeric. In addition, for the convenience of the compiler implementation, it would be concise to define one special type to represent the end of the token sequence (just like a string ends with '\0') to make the program concise. Let's allow the tokens to be a concatenated list connected by pointers so that we can handle input of arbitrary length.

It is rather long, but the compiler of the improved version with the tokenizer is shown below.

```
#include < ctype.h>.
#include < stdarg.h>.
#include < stdbool.h>.
#include < stdio.h>.
#include < stdlib.h>.
#include < string.h>.

// Token type
typedef enum {
 TK_RESERVED, // symbol
 TK_NUM,   //integer token
 TK_EOF,   // token representing the end of the input
} TokenKind;

typedef struct Token Token;

// Token type
struct Token {
 TokenKind kind; // token type
 Token *next;   // next input token
 int val;   // if kind is TK_NUM, the number
 char *str;   //token string
};

// Tokens we are currently focusing on
Token *token;

// Function to report an error
// Takes the same arguments as printf
void error(char *fmt, ...) {
 va_list ap;
 va_start(ap, fmt);
 vfprintf(stderr, fmt, ap);
 fprintf(stderr, "\n");
 exit(1);
}

// When the next token is the expected symbol, read the token forward one
// Return true. Otherwise, return false.
bool consume(char op) {
 if (token-> kind ! = TK_RESERVED || token->str[0] ! = op)
 return false;
 token = token-> next;
 return true;
}

// When the next token is the expected symbol, read the token one more time.
// Otherwise, report an error.
void expect(char op) {
 if (token-> kind ! = TK_RESERVED || token->str[0] ! = op)
 error("Not '%c'", op);
 token = token-> next;
}

// If the next token is a number, read the token forward by one and return
that number.
// Otherwise, report an error.
int expect_number() {
 if (token-> kind ! = TK_NUM)
     error("Not a number");
 int val = token-> val;
 token = token-> next;
 return val;
}

bool at_eof() {
 return token->kind == TK_EOF;
}

// Create a new token and connect it to cur
Token *new_token(TokenKind kind, Token *cur, char *str) {
 Token *tok = calloc(1, sizeof(Token));
 tok->kind = kind;
 tok->str = str;
 cur->next = tok;
 return tok;
}

// tokenize the input string p and return it
Token *tokenize(char *p) {
 Token head;
 head.next = NULL;
 Token *cur = & head;

 while (*p) {
 // Skip whitespace characters
 if (isspace(*p)) {
       p++;
 Continue;
     }

 if (*p == '+' || *p == '-') {
       cur = new_token(TK_RESERVED, cur, p++);
 Continue;
     }
 if (isdigit(*p)) {
       cur = new_token(TK_NUM, cur, p);
       cur-> val = strtol(p, &p, 10);
 Continue;
     }

     error("Cannot tokenize");
 }

 new_token(TK_EOF, cur, p);
 return head.next;
}

int main(int argc, char **argv) {
 if (argc ! = 2) {
     error("The number of arguments is incorrect. ");
 return 1;
 }

 // Tokenize.
 token = tokenize(argv[1]);

 // Output the first half of the assembly
 printf(". intel_syntax noprefix\n");
 printf(". globl main\n");
 printf("main:\n");

 // The first part of the expression must be a number, so check that
 // output the first mov instruction
 printf(" mov rax, %d\n", expect_number());

 // while consuming a sequence of `+ < number>` or `- < number>` tokens.
 // Output the assembly
 while (! at_eof()) {
 if (consume('+')) {
 printf(" add rax, %d\n", expect_number());
 Continue;
     }

 expect('-');
 printf(" sub rax, %d\n", expect_number());
 }

 printf(" ret\n");
 return 0;
}
```

The code is not very short, about 150 lines, but it is not too tricky, so you should be able to read it if you start from the top.

Let's discuss some of the programming techniques used in the code above.

- The token sequence to be read by the parser is represented by the global variable token. The parser reads the input by tracing the tokens in a concatenated list. This programming style of using global variables may not look pretty. However, in reality, it is often easier to read the parser code if the input token sequence is treated as a stream like standard input, as we are doing here. Therefore, we have adopted such a style here.
- The code that directly touches the token is divided into functions such as consume and expect, while the other functions do not directly touch the token.
- The tokenize function constructs a concatenated list. When constructing a concatenated list, the code can be simplified by creating a dummy head element, connecting new elements to it, and returning head->next at the end. The memory allocated for the head element is mostly wasted in this way, but since the cost of allocating local variables is almost zero, there is no need to worry about it.
- calloc is a function that allocates memory just like malloc, but unlike malloc, calloc clears the allocated memory to zero. Unlike malloc, calloc clears the allocated memory to zero. We decided to use calloc here to save the trouble of clearing the elements to zero.

This improved version should now be able to skip whitespace, so let's add the following one line test to test.sh.

```
assert 41 " 12 + 34 - 5 "
```

The exit code of a Unix process is supposed to be a number between 0 and 255, so when writing tests, make sure that the result of the entire expression fits between 0 and 255.

Add the test file to the git repository and this step is complete.

### Reference implementation

- ef6d1791eb2a5ef3

Step 4: Improve error messages

With the compiler we have built so far, if the input is grammatically wrong, we only know that there is an error somewhere. Let's try to improve that problem in this step. Specifically, we will be able to display intuitive error messages like the one below.

```
$ . /9cc "1+3++" > tmp.s
1+3++
     ^ It's not a number.

$ . /9cc "1 + foo + 5" > tmp.s
1 + foo + 5
     ^ Cannot tokenize.
```

In order to display such an error message, we need to be able to know how many bytes of input it is when an error occurs. To do so, let's decide to store the entire program string in a variable called `user_input`, and define a new error display function that receives a pointer that points to the middle of the string. The code is shown below.

```
// Input program
char *user_input;

// Report the error location.
void error_at(char *loc, char *fmt, ...) {
 va_list ap;
 va_start(ap, fmt);

 int pos = loc - user_input;
 fprintf(stderr, "%s\n", user_input);
 fprintf(stderr, "%*s", pos, " "); // print pos spaces
 fprintf(stderr, "^ ");
 vfprintf(stderr, fmt, ap);
 fprintf(stderr, "\n");
 exit(1);
}
```

The pointer received by `error_at` is a pointer pointing to the middle of the string that represents the entire input. If you take the difference between that pointer and the pointer pointing to the beginning of the input, you will know how many bytes of the input the error is in, and you can mark the location prominently with ^.

This step is completed if you make sure that `argv[1]` is stored in `user_input` and update the code like `error("Not a number")` to `error_at(token->str, "Not a number″)`.

If you have a working compiler, you should also write tests for the behavior when there is an error in the input, but for now, error messages are just output to help debugging, so you don't need to write any tests at this stage.

### Reference implementation

- c6ff1d98a1419e69

### Column: Source Code Formatter

In the same way that Japanese sentences with many errors in punctuation and orthography are unreadable, source code with incorrect indentation, inconsistent spaces, etc., cannot be called clean code even before the content of the source code. Be careful to write code that can be read without distraction by mechanically applying certain rules to the so-called unimportant parts such as code formatting.

When developing with several people, you need to discuss and decide what format to use, but in this book, since I am developing alone, I can choose any format I like from among some major formats.

Some recently developed languages provide official language formatters in order to eliminate the need for the controversial, but not essential, discussion of what format to choose. For example, the Go language has a command called gofmt that will format your source code nicely. gofmt does not have an option to choose the formatting style, and can only format to the only "official Go format" so to speak. By not giving you a choice, Go has completely solved the problem of how to format.

In C and C++, there is a formatter called clang-format, but in this book, we do not particularly want to recommend using such a tool. Try to be careful to write consistent looking code from the beginning, rather than writing oddly formatted code and formatting it later.

### Column: Security Bugs Caused by Indent Errors

I've had some serious security issues with iOS and macOS due to wrong indentation of the source code. The code where the bug was found is shown below.

```
if ((err = ReadyHash(&SSLHashSHA1, & hashCtx)) ! = 0)
 goto fail;
if ((err = SSLHashSHA1.update(& hashCtx, & clientRandom)) ! = 0)
 goto fail;
if ((err = SSLHashSHA1.update(& hashCtx, & serverRandom)) ! = 0)
```

```
 goto fail;
if ((err = SSLHashSHA1.update(& hashCtx, & signedParams)) ! = 0)
 goto fail;
 goto fail;
if ((err = SSLHashSHA1.final(& hashCtx, & hashOut)) ! = 0)
 goto fail;
```

Can the reader see where the bug is? This code looks like a normal piece of code, but if you look closely, you will see that the second goto statement from the bottom is not inside an if statement, but a goto statement that is always executed.

Unfortunately, this code was written in a function that validates TLS certificates, and as a result, most of the certificate validation code was unconditionally skipped in the goto statement, causing iOS/macOS to accept a bad certificate as a valid certificate (allowing spoofing of HTTPS sites). This bug was discovered and fixed in 2014. This bug was discovered and fixed in 2014. This bug is called the goto fail bug because of the fun wordplay of the extra goto fail causing the program to fail.

## Grammar description methods and recursive descent parsing

Now, the next step is to add multiplication, division, and precedence parentheses, i.e., `*`, `/`, `()`, to the language, but there is one major technical challenge to do so. This is because there is a rule that multiplication and division must be calculated first in an expression. For example, the expression `1+2*3` must be interpreted as `1+(2*3)`, but not as `(1+2)*3`. This rule of which operator should be ″attached″ first is called ″operator precedence".

How should I handle the precedence of the operators? The compiler I've built so far just reads the token sequence from the beginning and outputs the assembly, so if I just straightforwardly expand it as is and add `*` and `/`, it will compile `1+2*3` as `(1+2)*3`.

Existing compilers are naturally able to handle operator precedence very well. The compiler's parsing is so powerful that it can correctly interpret code of any complexity, as long as it follows the grammar. In fact, computers do not have the ability to read text like humans do, so parsing must be done only by some mechanical mechanism. What exactly is the mechanism that is working?

In this chapter, let's take a break from coding and learn about parsing techniques. In this chapter, we will discuss parsing techniques in the following order

1. Understand the final goal first by knowing the data structure of the parser output.
2. Learn the rules that define grammatical rules.
3. Learn techniques for writing parsers based on rules that define grammatical rules.

## Representation of grammatical structures in a tree structure

In parser implementations of programming languages, the input is usually a flat sequence of tokens, and the output is a tree representing a nested structure. The compiler in this book also follows this structure.

In the C language, grammatical elements such as `if` and `while can be` nested. The tree structure is a natural way to represent such things.

Mathematical expressions have structures such as calculating the inside of parentheses first, or calculating multiplication and division before addition and subtraction. These structures may not look like a tree at first glance, but in fact, a tree can be used to represent the structure of an expression in a very simple way. For example, the expression `1*(2+3) can be` represented by the following tree.

Tree representing `1*(2+3)`

If we adopt the rule of calculating in order from the end of the tree, the above tree represents the expression 1 multiplied by 2+3. In other words, in the above tree, the specific order of calculation of `1*(2+3)` is expressed in the shape of the tree itself.

Here is another example. The tree below represents `7-3-3`.

Tree representing `7-3-3`

In the above tree, the result of the application of the rule "subtraction must be calculated from left to right" is explicitly expressed as the shape of the tree. In other words, the above tree represents the expression `(7-3)-3 = 1`, not the expression 7-(3-3) = 7. If it were the latter expression, then the tree representing it would be deeper to the right, not to the left.

Operators that must be computed from the left are called "left-bound" operators, and those that must be computed from the right are called "right-bound" operators. in C, most operators are defined as left-bound, except for = in assignments.

In a tree structure, you can represent any number of long expressions by making the tree deeper. The following tree is a tree representing 1*2+3*4*5.

Tree representing 1*2+3*4*5

The above kind of tree is called a "syntax tree". In particular, a syntax tree in which redundant elements such as parentheses for grouping are not left in the tree, but are expressed as compactly as possible, is called an abstract syntax tree (AST). All of the above syntax trees can be considered abstract syntax trees.

The abstract syntax tree is an internal representation of the compiler, so it can be defined as appropriate for the implementation. However, arithmetic operators such as addition and multiplication are defined as operations on two sides, the left side and the right side, so it is natural to use a binary tree in any compiler. On the other hand, expressions in the body of a function, which can be any number of elements that are executed in sequence, are naturally represented by a tree with all child elements flat.

The goal in parsing is to construct an abstract syntax tree. The compiler will first parse and convert the input token sequence into an abstract syntax tree, and the next step is to convert the syntax tree into an assembly.

## Defining grammars with generative rules

Now, let's learn how to describe the syntax of a programming language. Most of the syntax of a programming language is defined using something called a "production rule". A production rule is a rule that defines a grammar recursively.

Let's think about natural language for a moment. In Japanese, grammar is a nested structure. For example, if you replace the noun "flower" with the noun phrase "red flower" in the sentence "the flower is beautiful," you get the correct sentence. You can also include "I thought the little red flower was beautiful" in another sentence.

Let's think of these grammars as being defined by rules such as "a sentence consists of a subject and a predicate" or "a noun phrase consists of a noun or an adjective followed by a noun phrase". Then, by starting with a "sentence" and developing it according to the rules, you can create countless valid sentences in the defined grammar.

Or conversely, for a sentence that already exists, we can consider whether the string has any structure by considering the expansion procedure that matches it.

Originally, the above ideas were invented for natural languages, but since they have a very high affinity with the data handled by computers, the generation rules are used in various parts of computers, including programming languages.

**Column: Chomsky's Generative Grammar**

It was a linguist named Noam Chomsky who came up with the idea of generative grammar. His ideas have had a huge impact on linguistics and computer science.

According to Chomsky's hypothesis, the reason why humans can speak language is that we are born with a dedicated circuit in our brain for acquiring generative rules. Humans have the ability to acquire language rules recursively, which is why we can speak language. Non-human animals do not have the ability to acquire language, but he believed that this is because the circuits for acquiring generative rules do not exist in the brains of non-human animals. Chomsky's argument is still considered quite convincing, although it has not been proven or disproved in the nearly 60 years since the hypothesis was first published.

## Description of generation rules by BNF

One notation for describing generation rules in a compact and easy-to-understand way is BNF (Backus-Naur form) and its extension, EBNF (Extended BNF). In this book, we will use EBNF to explain the grammar of C. In this section, we will first explain BNF, and then the extended part of EBNF.

In BNF, each generating rule is expressed in the form $A = \alpha_1\alpha_2\cdots$, meaning that the symbol A can be expanded to $\alpha_1\alpha_2\cdots$.This means that the symbol `A` can be expanded to

$\alpha_1\alpha_2\cdots$, where $\alpha_1\alpha_2\cdots$ is a sequence of zero or more symbols, and can contain both symbols that cannot be further expanded, and symbols that can be further expanded (coming to the left side in any of the generation rules).

A symbol that cannot be expanded any further is called a "terminal symbol," while a symbol that is on the left side of one of the generation rules and can be expanded is called a "nonterminal symbol. A grammar defined by such generation rules is generally called a "context free grammar".

A non-terminal symbol may match more than one generation rule. For example, if there are both rules `A = `$\alpha_1$ and `A = `$\alpha_2$, it means that `A can` be expanded to either $\alpha_1$ or $\alpha_2$.

The right-hand side of the generation rule can be empty. In such a rule, the symbols on the left-hand side will be expanded into a sequence of zero length symbols (i.e., nothing). However, for display purposes, omitting the right-hand side makes it difficult to understand the meaning, so in such a case, the usual BNF rule is to write epsilon on the right-hand side to indicate that there is nothing. This rule is also used in this book.

The string should be enclosed in double quotes and written as `"foo"`. Strings are always terminated.

The above are the basic BNF rules; in EBNF, in addition to the BNF rules, complex rules can be concisely written down using the following symbols

| way of writing | Meaning |
|---|---|
| `A*` | 0 or more repetitions of `A` |
| `A?` | `A` or ε |
| `A \| B` | `A` or `B` |
| `( ... )` | grouping |

For example, with A = ("fizz" | "buzz")*, A is the string in which "fizz" or "buzz" is repeated zero or more times, i.e.

- ""
- "fizz."
- "buzz."
- "fizzfizz."
- "fizzbuzz".
- "buzzfizz."
- "buzzbuzz."
- "fizzfizzfizz."
- "fizzfizzbuzz."
- …

You can expand it to any of the following

### Column: BNF and EBNF

For a normal BNF that is not Extended, *,? and|, ( ...). ), but the statements that can be generated with BNF are the same as those that can be generated with EBNF. This is because EBNF can be converted to BNF by rewriting as follows

| **EBNF** | Corresponding **BNF** |
|---|---|
| A = α* | A = αA and A = ε |
| A = α? | A = α and A = ε |
| A = α \| β | A = α and A = β |
| A = α $(\beta_1\beta_2\cdots)$ γ | A = α B γ and B = $\beta_1\beta_2\cdots$ |

For example, when generating the sentence ααα from A using the generation rules A = αA and A = ε, the order of expansion is A → αA → ααA → αααA → αααα.

As you can see, the * and ? Notations such as * and ? are just shortcuts, but short notations are more desirable and easier to understand, so if a short notation is available, it is usually used for a concise description.

## Simple generation rules

As an example of writing a grammar using EBNF, consider the following generation rule.

```
expr = num ("+" num | "-" num)*
```

We assume that num is defined separately somewhere as a symbol representing a number. In this grammar, expr means that there is first one num, followed by zero or more "+ and num, or - and num". This rule actually represents the grammar of addition and subtraction expressions.

Starting from expr and expanding it, you can create arbitrary strings of addition and subtraction, such as 1, 10+5, or 42-30+2. Check out the result of the expansion below.

```
expr → num → "1"

expr → num "+" num
      → "10" "+" "5"

expr → num "-" num "+" num
      → "42," "-," "30," "+," "2."
```

This kind of expansion procedure can be represented not only by arrows for each order of expansion, but also by a tree structure. The syntax tree for the above expression is shown below.

1 syntax tree 10+5 syntax trees 42-30+2 syntax trees

By using a tree structure, it is easier to see which non-terminal symbols are expanded into which symbols.

A syntax tree that contains all the tokens in the input and is a perfect one-to-one match to the grammar, as shown in the figure above, is sometimes called a "concrete syntax tree". This term is often used when you want to contrast it with an abstract syntax tree.

Note that in the above concrete syntax tree, the rule that addition and subtraction are calculated from the left is not expressed in the form of a tree. In the grammar described here, such a rule is not expressed using EBNF, but rather written as a proviso in the language specification as "addition and subtraction are calculated from the left first". The parser would take both EBNF and the proviso into account, read the token sequence representing the expression, and construct an abstract syntax tree that properly represents the order of evaluation of the expression.

Therefore, in the above grammar, the concrete syntax tree represented by EBNF and the abstract syntax tree that is the output of the parser only roughly match in shape. It is possible to define a grammar so that the abstract syntax tree and the concrete syntax tree have the same structure as much as possible, but then the grammar becomes redundant and it becomes difficult to understand how to write the parser. A grammar like the one above is an easy to handle way of expressing a grammar that strikes a good balance between the rigor of formal grammar description and the clarity of natural language supplementation.

## Expression of operator precedence by generation rules

Generative rules are a very powerful tool for expressing grammars. The precedence of operators can also be expressed in the generation rules by devising a grammar. The grammar is shown below.

```
expr = mul ("+" mul | "-" mul)*
mul = num ("*" num | "/" num)*
```

In the previous rule, `expr` was directly expanded into `num`, but now the rule is that `expr` is expanded into `num` via `mul`, where `mul` is the generation rule for multiplication and division, and `expr`, which performs addition and subtraction, uses `mul` as a component, so to speak. In this grammar, the rule that multiplication and division are attached first is naturally expressed in the syntax tree. Let's take a look at some concrete examples.

1*2+3 syntax tree1+2*3 syntax tree1*2+3*4*5 syntax tree

In the above tree structure, multiplication always appears toward the end of the tree rather than addition. In fact, since there is no rule to go back from `mul` to `expr`, there is no way to create a tree with addition under multiplication, but even so, it is quite strange that such a simple rule can express priority so well as a tree structure. Readers are encouraged to check the correctness of the syntax tree by actually comparing the generation rules with the syntax tree.

## Generation rules with recursion

In generative grammars, recursive grammars can also be written in the usual way. Below is the generative rule for a grammar with precedence brackets added to the four operations.

```
expr    = mul ("+" mul | "-" mul)*
mul = primary ("*" primary | "/" primary)*
primary = num | "("expr ")""
```

If you compare the above grammar with the previous one, you will see that the primary, i.e., num or "(" expr ")", is allowed where num was allowed before. In other words, in this new grammar, expressions enclosed in parentheses will be treated with the same "stickiness" as a single number. Let's look at an example.

The next tree is a 1*2 syntax tree.

1*2 syntax tree

The next tree is a 1*(2+3) parse tree.

1*(2+3) parse tree

Comparing the two trees, we can see that only the result of the expansion of the primary of the right branch of mul is different. The rule that "primary" that appears at the end of the expansion result can be expanded to a single number or to any expression enclosed in parentheses is reflected in the tree structure. Isn't it a bit impressive that such a simple generation rule can also handle the precedence of parentheses?

## recursive descent parsing

Given a set of generation rules for the C language, we can mechanically generate arbitrary C programs that are correct from the point of view of the generation rules by expanding them as we go. However, what we want to do in 9cc is rather the opposite. What we want to do in 9cc, however, is rather the opposite: we want to know the structure of a syntax tree in which a C program given as a string from the outside is expanded to become the string of the input, that is, the same string as the input.

As a matter of fact, for certain generation rules, given a rule, it is possible to mechanically write code to find a syntax tree that matches the sentence generated by the rule. The "recursive descent parsing method" described here is one such technique.

As an example, let's consider the grammar of the four arithmetic operations. Let us restate the grammar of the four arithmetic operations.

```
expr     = mul ("+" mul | "-" mul)*
mul = primary ("*" primary | "/" primary)*
primary = num | "("expr ")""
```

The basic strategy when writing a parser in recursive descent parsing is to map each of these non-terminal symbols directly into a function. Thus, the parser will have three functions: `expr`, `mul`, and `primary`. Each function parses the token sequence as its name implies.

Let's consider this in concrete code. The input passed to the parser is a sequence of tokens. We want to create and return an abstract syntax tree from the parser, so let's define the node types of the abstract syntax tree. The node types are shown below.

```
// Types of nodes in abstract syntax trees
typedef enum {
 ND_ADD, // +
 ND_SUB, // -
 ND_MUL, // *
 ND_DIV, // /
 ND_NUM, // integer
} NodeKind;

typedef struct Node Node;

// Type of the node in the abstract syntax tree
struct Node {
 NodeKind kind; // type of the node
 Node *lhs;    // left side
 Node *rhs;    // right side
 int val;    // use only if kind is ND_NUM
};
```

The words `lhs` and `rhs mean` left-hand side and right-hand side, respectively.

We will also define a function to create a new node. Since there are two types of arithmetic operations in this grammar: binary operators that take a left side and a right side, and numbers, we will prepare two functions for these two types.

```
Node *new_node(NodeKind kind, Node *lhs, Node *rhs) {
 Node *node = calloc(1, sizeof(Node));
 node->kind = kind;
 node-> lhs = lhs;
 node-> rhs = rhs;
 return node;
}

Node *new_node_num(int val) {
 Node *node = calloc(1, sizeof(Node));
 node->kind = ND_NUM;
 node-> val = val;
 return node;
}
```

Now, let's write a parser using these functions and data types. The + and - are supposed to be left join operators. The function to parse the left join operator is written as a pattern as follows

```
Node *expr() {
 Node *node = mul();

 for (;;) {
 if (consume('+'))
       node = new_node(ND_ADD, node, mul());
 else if (consume('-'))
       node = new_node(ND_SUB, node, mul());
 else
 return node;
  }
}
```

consume is the function we defined in the previous step, which reads the input one token forward and returns true when the next token in the input stream matches the argument.

If you read the expr function carefully, you will see that the generation rule expr = mul ("+" mul | "-" mul)* is directly mapped to function calls and loops. In the abstract syntax tree returned from the expr function above, the operators are left-joined, that is, the left branch of the returned node is deeper.

Let's also define the mul function, which is used by the expr function. Since * and / are also left join operators, they can be written in the same pattern. The function is shown below.

```
Node *mul() {
 Node *node = primary();

 for (;;) {
 if (consume('*'))
       node = new_node(ND_MUL, node, primary());
 else if (consume('/'))
       node = new_node(ND_DIV, node, primary());
 else
 return node;
 }
}
```

The function call relationship in the above code directly corresponds to the generation rule mul = primary ("*" primary | "/" primary)*.

Finally, let's define the primary function. primary does not read left join operators, so the code will not follow the above pattern, but by mapping the generation rule primary = "(" expr ")" | num directly to the function call, the primary function can be written as follows.

```
Node *primary() {
 // If the next token is "(", then it should be "(" expr ")"
 if (consume('(')) {
     Node *node = expr();
 expect(')');
 return node;
 }

 // otherwise it should be a number
 return new_node_num(expect_number());
}
```

Now that we have all the functions in place, can we really parse the token sequence with them? At first glance, it may not look like it, but with this set of functions, you can parse the token sequence properly. As an example, let's consider the expression 1+2*3.

The first thing that is called is expr. We assume that the expression as a whole is an expr (and in this case, it really is), and start reading the input. Then the function call is made from expr→mul→primary, the token 1 is read, and expr returns a syntax tree representing 1 as the return value.

Next, the expression consume('+') in expr becomes true, so the token + is consumed and mul is called again. The remaining input at this stage is 2*3.

The expression consume('*') in mul becomes true, so mul calls primary again and reads the token 3. The result is 2*3. As a result, mul will return a syntax tree representing 2*3.

In the expr that returns, the syntax tree representing 1 and the syntax tree representing 2*3 are combined to construct a syntax tree representing 1+2*3, which becomes the return value of expr. In other words, 1+2*3 has been correctly parsed.

The calling relationship of the functions and the tokens that each function reads are shown in the figure below. In the figure below, there is a layer of expr corresponding to the whole 1+2*3, which represents a call to expr that reads the whole input; above expr, there are two mul, which represent another mul call that reads 1 and 2*3.

Function call relationship when parsing 1+2*3

A more complex example is shown below. The figure below shows the calling relationship of the function when parsing 1*2+(3+4).

Function call relationship when parsing 1*2+(3+4)

For programmers who are not familiar with recursion, recursive functions like the one above may seem confusing. To be honest, even though I am very familiar with recursion, I feel that it is a kind of magic that this kind of code works. Recursive code is somewhat mysterious to me even when I know how it works, but perhaps that is the way it is. Try tracing the code in your mind often, many times, and make sure that the code works properly.

The parsing technique of mapping one generation rule to one function as described above is called "recursive descent parsing". In the above parser, only one token was looked ahead to determine which function to call or return. A recursive descent parser that looks ahead to only one token is called an LL(1) parser. A recursive descending parser that reads only one token ahead is called an LL(1) parser, and a grammar that can be written by an LL(1) parser is called an LL(1) grammar.

stack machine

In the previous chapter, we described an algorithm for converting a token sequence into an abstract syntax tree. By choosing a grammar that takes into account the precedence of operators, we can now create an abstract syntax tree where * and / always come at the end of the branch, compared to + and -. But how do we convert this tree into an assembly? In this chapter we will explain how.

First, let's consider why we cannot convert to assembly in the same way as addition and subtraction. In compilers that can do addition and subtraction, RAX was used as a register of results, and additions and subtractions were performed there. In other words, the compiled program kept only one intermediate calculation result.

However, when multiplication and division are involved, it is not always possible to have only one intermediate calculation result. As an example, consider 2*3+4*5. In order to perform addition, both sides must have been calculated, so we need to calculate 2*3 and 4*5 before adding. In other words, in this case, we need to be able to keep two results of the calculation in the middle to be able to calculate the whole thing.

A computer called a "stack machine" makes it easy to compute these kinds of things. Let's take a break from the abstract syntax tree created by the parser and learn about stack machines.

## Concept of Stack Machine

A stacked machine is a computer that has a stack as its data storage area. Therefore, there are two basic operations in a stack machine: "push to stack" and "pop from stack". Pushing puts a new element at the top of the stack. Popping removes an element from the top of the stack.

Arithmetic instructions in a stack machine act on the elements on the top of the stack. For example, the ADD instruction in a stack machine pops two elements from the top of the stack, adds them together, and pushes the result onto the stack (to avoid confusion with x86-64 instructions, all instructions in virtual stack machines are written in uppercase). To put it another way, ADD is an instruction that replaces two elements at the top of the stack with one element of the result of adding them together.

The SUB, MUL, and DIV instructions, like ADD, are instructions that replace two elements at the top of the stack with one element that subtracts, multiplies, or divides them.

The PUSH instruction assumes that the argument elements are piled up on the top of the stack. Although not used here, we can also consider the POP instruction, which removes an element from the top of the stack and discards it.

Now, let's consider using these instructions to compute 2*3+4*5. Using the stack machine defined above, we should be able to calculate 2*3+4*5 with the following code.

```
// Calculate 2*3
PUSH 2
PUSH 3
MUL

// Calculate 4*5
PUSH 4
PUSH 5
MUL

// Calculate 2*3 + 4*5
ADD
```

Let's take a closer look at this code. We'll assume that the stack contains some value in advance. The stack is assumed to be pre-populated with some value, which is not important here, so we'll show it as "…". The stack is assumed to extend from top to bottom in the diagram.

The first two pushes pushes 2 and 3 onto the stack, so by the time the immediate MUL is executed, the state of the stack will look like this

| |
|---|
| … |
| 2 |
| 3 |

MUL removes the two values at the top of the stack, i.e., 3 and 2, and pushes the result of multiplying them, i.e., 6, onto the stack. Thus, after the execution of MUL, the state of the stack will look like this

| |
|---|
| ... |
| 6 |

Next, PUSH pushes 4 and 5, so just before the second MUL is executed, the stack should look like this

| |
|---|
| ... |
| 6 |
| 4 |
| 5 |

Now, when we run MUL, 5 and 4 are removed and replaced by 20, the result of multiplying them. Thus, after running MUL, we get

| |
|---|
| ... |
| 6 |
| 20 |

Notice that the results of the calculation of 2*3 and 4*5 are well placed on the stack. If we run ADD in this state, 20+6 will be computed and the result will be pushed onto the stack, so the stack should eventually be in the following state

| |
|---|
| ... |
| 26 |

If we assume that the result of the stack machine calculation is the value left at the top of the stack, then 26 is the result of 2*3+4*5, which means that we have properly calculated that expression.

The stack machine can calculate not only this expression, but any expression that has multiple intermediate results. With the stack machine, any sub-expression can be compiled successfully in the above way, as long as it keeps the promise of leaving one element on the stack as the result of executing it.

### Column: CISC and RISC

The x86-64 is a typical ″CISC" style processor, with an instruction set that has progressively evolved from the 8086 released in 1978. x86-64 processors are characterized by the fact that machine language operations can take memory addresses as well as registers, the CISC processors are characterized by the fact that machine language operations can take memory addresses instead of only registers, that machine language instructions are variable in length, and that they have many instructions that perform complex operations in a single instruction, which is useful for assembly programmers.

RISC was invented in the 1980s as opposed to CISC. RISC processors are characterized by the fact that operations are always performed only between registers, and that the only operations on memory are loads into and stores from registers. The length of machine language instructions is the same for all instructions, and it does not have complex instructions, which are useful for assembly programmers, but only simple instructions generated by the compiler.

The x86-64 is one of the few survivors of CISC, and almost all major processors other than the x86-64 are based on RISC. Specifically, ARM, PowerPC, SPARC, MIPS, and RISC-V (Risk Five) are all RISC processors.

RISC does not have operations between memory and registers like x86-64. There is no register aliasing. There is no rule that a particular integer register is used in a particular way in a particular instruction. The x86-64 instruction set looks old-fashioned to modern eyes, where such instruction sets have become mainstream.

RISC processors dominated the processor industry because their simple design made them easy to speed up. So why did the x86-64 succeed in surviving? There was a huge market need for a fast x86 processor that could take advantage of existing software assets, and Intel and Intel-compatible chip makers innovated to meet this need. x86 instructions are internally converted into a kind of RISC instructions by the CPU's instruction decoder. Intel internally converted x86 instructions into some sort of RISC instructions in the CPU's instruction decoder, making the x86 an internal RISC processor. This made it possible to apply the same techniques that had made RISC so fast to the x86.

## Compiling to a stack machine

This section describes how to convert an abstract syntax tree into stack machine code. If you can do that, you will be able to parse an expression consisting of four arithmetic operations, construct an abstract syntax tree, and compile it to a stack machine using x86-64 instructions for execution. In other words, you will be able to write a compiler that can perform the four arithmetic operations.

In the stack machine, when you computed a partial expression, one value of the result, whatever it was, would remain at the top of the stack. For example, consider a tree like the one below.

Abstract syntax tree for addition

A and B are abstractions of a subtree, which actually means a node of some type. However, the specific type or the shape of the tree is not important when compiling the whole tree. When compiling this tree, you can do the following

1. Compile the left partial tree
2. Compile the right partial tree
3. Output code to replace the two values in the stack with the result of adding them.

After executing the code in 1, whatever the specific code is, there should be a single value on the top of the stack that represents the result of the left subtree. Similarly, after executing the code in 2, there should be a single value on the top of the stack that represents the result of the right subtree. Therefore, to calculate the value of the whole tree, we just need to replace those two values by their sum.

Thus, when we compile an abstract syntax tree into a stack machine, we think recursively and output more and more assemblies as we go down the tree. This may seem somewhat difficult to readers who are not familiar with the idea of recursion, but recursion is a standard technique when dealing with self-similar data structures such as trees.

Let's consider the following example in detail.

Abstract syntax tree for addition and multiplication

The function that does the code generation takes the root node of the tree.

Following the above procedure, the first thing the function does is to compile the left subtree, which compiles the number 2. Since the result of calculating 2 is still 2, the result of compiling that subtree is PUSH 2.

The code generation function then tries to compile the right subtree. This will recursively compile the left side of the subtree, resulting in the output PUSH 3. The next step is to compile the right side of the subtree, which results in PUSH 4.

After that, the code-generating function will go back to the original recursive call and output the code according to the type of the operator of the partial tree. The first output is the code that replaces the two elements at the top of the stack by multiplying them together. The next output is the code that replaces the two elements at the top of the stack with the sum of the two elements. The resulting output will be the assembly below.

```
PUSH 2
PUSH 3
PUSH 4
MUL
ADD
```

Using such a method, we can mechanically reduce the abstract syntax tree to an assembly.

## How to realize stacked machines in x86-64

So far, we have been talking about a virtual stack machine. The actual x86-64 is not a stack machine but a register machine. x86-64 operations are usually defined for operations between two registers, not for operations on two values at the top of the stack. Therefore, in order to use the stack machine technique with x86-64, you need to emulate the stack machine in a sense with a register machine.

It is relatively easy to emulate a stack machine in a register machine. All you have to do is implement what is a single instruction in the stack machine using multiple instructions.

Let me explain a specific method for doing so.

The first step is to prepare a register that points to the top element of the stack. That register is called the stack pointer. If you want to pop the two values at the top of the stack, take out the two elements pointed to by the stack pointer, and change the stack pointer by the amount of the removed elements. In the same way, when you want to push, just change the value of the stack pointer and write it to the memory area it points to.

The x86-64 RSP register is designed to be used as a stack pointer. x86-64 instructions such as `push` and `pop implicitly` use the RSP as a stack pointer to access the memory pointed to by the RSP while changing its value. Therefore, when using the x86-64 instruction set like a stack machine, it is straightforward to use the RSP as a stack pointer. Now, let's compile the expression `1+2 using` x86-64 as a stack machine. The x86-64 assembly is shown below.

```
// Push left side and right side
push 1
push 2

// Pop the left and right sides into RAX and RDI and add them.
pop rdi
pop rax
add rax, rdi

// Push the result of the addition onto the stack
push rax
```

Since there is no ″add the two elements pointed to by the RSP" instruction in x86-64, it is necessary to load them into a register, perform the addition, and then push the result back onto the stack. This is what the `add` instruction above does.

Similarly, if we implement `2*3+4*5` on x86-64, we get the following

```
// Calculate 2*3 and push the result to the stack
push 2
push 3

pop rdi
pop rax
mul rax, rdi
push rax

// Calculate 4*5 and push the result to the stack
push 4
push 5

pop rdi
pop rax
mul rax, rdi
push rax

// Add the two values at the top of the stack
// i.e. calculate 2*3+4*5
pop rdi
pop rax
add rax, rdi
push rax
```

As you can see, with the stack manipulation instructions of x86-64, you can run code that is quite similar to a stack machine, even on x86-64.

The following `gen` function is a direct implementation of this technique in a C function.

```
void gen(Node *node) {
 if (node->kind == ND_NUM) {
 printf(" push %d\n", node->val);
 return;
 }

 gen(node-> lhs);
 gen(node-> rhs);

 printf(" pop rdi\n");
 printf(" pop rax\n");

 switch (node->kind) {
 case ND_ADD:
 printf(" add rax, rdi\n");
 break;
 case ND_SUB:
 printf(" sub rax, rdi\n");
 break;
 case ND_MUL:
 printf(" imul rax, rdi\n");
 break;
 case ND_DIV:
 printf(" cqo\n");
 printf(" idiv rdi\n");
 break;
 }

 printf(" push rax\n");
}
```

Although it is not a particularly important point in parsing or code generation, the idiv instruction with a tricky specification is used in the above code, so let's discuss it.

idiv is an instruction that performs signed division. idiv in x86-64 has a straightforward specification; if idiv in x86-64 had been straightforward, the above code would have originally been written as idiv rax, rdi, but such a division instruction that takes two registers does not exist in x86-64. Instead, idiv implicitly takes RDX and RAX, considers the sum of the two as a 128-bit integer, divides it by the 64-bit value in the argument register, and sets the quotient in RAX and the remainder in RDX. In the above code, cqo is called before idiv is called because the cqo instruction can extend the 64-bit value in RAX to 128 bits and set it in RDX and RAX.

This concludes our discussion of stack machines. By reading this far, the reader should now be able to parse complex syntax and reduce the resulting abstract syntax tree to machine code. To put that knowledge to use, let's get back to the task of creating a compiler!

### Column: Optimizing Compilers

The x86-64 assembly used by the author in this chapter may seem quite inefficient. For example, an instruction that pushes a number onto the stack and pops it should be written as an instruction that moves the value directly into a register, which would require only one instruction. Some readers may be tempted to optimize such assemblies by removing redundancy from them. However, please do not give in to that temptation. In the very first code generation, it is desirable to output redundant code in favor of ease of implementation by the compiler.

You can add optimization paths to 9cc later if necessary. It is not difficult to re-scan the generated assembly and replace an instruction sequence that appears in a particular pattern with another instruction sequence. For example, if you make a rule such as "replace pop immediately after push with mov" or "if successive adds add immediate values to the same register, replace them with a single add that adds the sum of the immediate values" and apply it mechanically, you can turn redundant code into more efficient code without changing its meaning. If the rule is applied mechanically, redundant code can be replaced with more efficient code without changing the meaning.

If you mix code generation and optimization, the compiler becomes more complex. If the code is difficult from the start, it is rather difficult to add optimization paths later on; as Donald Knuth said, "Premature optimization is the bane of all evil. As Donald Knuth said, "premature optimization is the bane of all evil", so please make sure that the compiler you write only considers ease of implementation. Don't worry about obvious redundancies in the output, as they can be removed later.

## Step 5: Create a language that can do the four arithmetic operations.

In this chapter, we will modify and extend the compiler we have created in the previous chapters to handle expressions of the four arithmetic operations including precedence brackets. Since we have all the necessary parts, there is only a small amount of new code to write. Try to change the main function of the compiler to use the newly created parser and code generator. Your code should look like the one below.

```
int main(int argc, char **argv) {
 if (argc ! = 2) {
     error("The number of arguments is incorrect. ");
 return 1;
 }

 // Tokenize and parse.
 user_input = argv[1];
 token = tokenize(user_input);
 Node *node = expr();

 // Output the first half of the assembly
 printf(". intel_syntax noprefix\n");
 printf(". globl main\n");
 printf("main:\n");
```

```
  // Code generation while descending the abstract syntax tree
  gen(node);

  // Since the value of the entire expression should remain at the top of the stack
  // Load it into RAX and make it the return value from the function
  printf(" pop rax\n");
  printf(" ret\n");
  return 0;
}
```

Having progressed to this stage, the expression consisting of addition, subtraction, multiplication, and precedence parentheses should now compile correctly. Let's add some tests.

```
assert 47 '5+6*7'
assert 15 '5*(9-6)'
assert 4 '(3+5)/2'
```

Up to this point, for the sake of explanation, we have been talking as if we were implementing `*`, `/`, and `()` all at once, but in reality, it is better to avoid implementing them all at once. Originally, you had the ability to add and subtract, so try to introduce abstract syntax trees and code generators that use them without breaking that functionality first. No new tests are needed at that time, since we are not adding any new functionality. After that, please implement `*`, `/`, and `()` with tests included.

### Reference implementation

- 3c1e3831009edff2

### Column: Memory Management in 9cc

Having read this book so far, the reader may be wondering how memory management works in this compiler. The code so far uses calloc (a variant of malloc), but does not call free. This means that the allocated memory is not released. Isn't this a bit tricky at all?

In fact, this design of ″no memory management as a memory management policy″ is a deliberate choice by the author after considering various trade-offs.

One of the advantages of this design is that by not releasing memory, you can write code as if it were a language with a garbage collector. This not only eliminates the need to write memory management code, but also eliminates the puzzling bugs associated with manual memory management.

On the other hand, the problems caused by not doing free are practically non-existent if you consider running it on a computer like a regular PC. The compiler is a short-lived program that only reads a single C file and outputs the assembly. All memory allocated at the end of the program is automatically freed by the OS. Therefore, the only question is how much memory to allocate in total, but according to my actual measurement, the memory usage is only about 100 MiB even when compiling a rather large C file. Therefore, not freeing is a realistically effective strategy. For example, the D language compiler DMD has adopted the same policy of only malloc and not free. [1]

## Step 6: Unary plus and unary minus

The - operator, which performs subtraction, can be written not only between two terms, as in 5-3, but also in front of a single term, as in -3. Similarly, the + operator can be written without the left-hand side, as in +3. Such an operator that takes only one term is called a ″unary operator". On the other hand, an operator that takes two terms is called a ″binary operator″.

In addition to + and -, C has other unary operators such as & to get a pointer and * to dereference a pointer, but in this step, we will implement only + and -.

The unary + and - are the same symbols as the binary + and -, but their definitions are different: the binary - is defined as an operation that subtracts the right side from the left side, while the unary - has no left side, so the definition of the binary - does not make sense. In C, the unary - is defined as an operation that reverses the positive or negative value of the right side. In C, unary - is defined as an operation that reverses the positive or negative value of the right-hand side. Unary + is an operator that returns the right-hand side as is. In C, unary- is defined as an operation that reverses the positive or negative value of the right-hand side.

It is appropriate to think of + and - as having multiple operators of the same name with similar but different definitions: unary and binary. Whether they are unary or binary is a matter of context. A new grammar that includes unary +/- would look like this

```
expr    = mul  ("+" mul | "-" mul)*
mul = unary ("*" unary | "/" unary)*
unary   = ("+" | "-")? primary
primary = num | "("expr ")""
```

The new grammar above has a new non-terminal symbol called unary, and mul uses unary instead of primary. unary = ("+" | "-")? In the rule "primary", the non-terminal symbol unary represents an element that may or may not have a single + or -, followed by a primary.

Please check that expressions like -3, -(3+5), and -3*+5 match this new grammar. The syntax tree for -3*+5 is shown below.

-3*+5 syntax tree

Let's modify the parser to follow this new grammar. As usual, we can map the grammar directly to a function call and the parser change should be complete.

```
Node *unary() {
 if (consume('+'))
 return primary();
 if (consume('-'))
 return new_node(ND_SUB, new_node_num(0), primary());
 return primary();
}
```

Here, we decided to replace +x with x and -x with 0-x in the parsing step. Therefore, no changes to the code generator are needed in this step.

Write a few tests and check them in along with the code to add the unary +/- and you are done with this step. When writing the tests, try to keep the test results in the range of 0-255. -Expressions like 10+20 use the unary - but the overall value is a positive number, so use this kind of test.

### Reference implementation

- bb5fe99dbad62c95

### Column: Unary Plus and Good or Bad Grammar

The unary + operator was not present in the original C compiler, but was officially added to the language when C was standardized by ANSI (American National Standards Institute) in 1989. As long as unary - exists, it is certainly better to have unary + as well, as it is more symmetric and better in that sense, but in reality, unary + has no particular use.

On the other hand, there is a side-effect of adding the unary + to the grammar: if someone new to C accidentally writes the += operator as i =+ 3. Without the unary +, this is just an invalid expression, but with the unary +, it will be interpreted as if you had written i = +3, and the compiler will silently accept it as a legitimate assignment expression to assign 3 to i.

ANSI's C standardization committee made the decision to add unary + to the language after understanding the above problems, what do you think? If you were on the C standardization committee at that time, would you agree or disagree? Would you oppose it?

## Step 7: Comparison operators

In this section, we will implement <, <=, >, >=, ==, and ! = are implemented in this section. Although these comparison operators seem to have special meanings, they are actually ordinary binary operators that take two integers and return one integer, just like + and -. Just as + returns the result of adding both sides, for example, == returns 1 if both sides are the same and 0 if they are different.

# Change the tokenizer.

All of the symbolic tokens we have dealt with so far have a length of one character, and we have assumed that in our code, but in order to handle comparison operators such as ==, we need to generalize our code. To be able to store the length of the string in a token, we will store a member called len in the Token structure. The type of the new structure is shown below.

```
struct Token {
 TokenKind kind; // token type
 Token *next;    // next input token
 int val;    // if kind is TK_NUM, the number
 char *str;    //token string
 int len;    //length of the token
};
```

With this change, we also need to modify the consume and expect functions so that they take strings instead of characters. An example of the changes we made is shown below.

```
bool consume(char *op) {
 if (token-> kind ! = TK_RESERVED ||
 strlen(op) ! = token-> len ||
 memcmp(token->str, op, token-> len))
 return false;
 token = token-> next;
 return true;
}
```

When tokenizing a symbol consisting of multiple characters, the longest token must be tokenized first. For example, if the rest of the string starts with >, if you check for the possibility that the string starts with > without first checking for the possibility that it is >=, as in strncmp(p, ">=", 2), then >= will be incorrectly tokenized as two tokens, > and =. If you check for the possibility that the token starts with > without first checking for the possibility that it is >=, then >= will be incorrectly tokenized as two tokens: > and =.

## New grammar

In order to add support for comparison operators to the parser, let's consider what the grammar would look like with the addition of comparison operators. If we write the operators we have seen so far in order of priority from lowest to highest, we get the following

1. == ! = !
2. < <= > >=
3. + -
4. * /
5. Unary + Unary -
6. ()

If we think of the grammar in the same way as expr and mul, the new grammar with the addition of the comparison operator would look like this

```
expr       = equality
equality   = relational ("==" relational | "! =" relational)*
relational = add ("<" add | "<=" add | ">" add | ">=" add)*
add        = mul ("+" mul | "-" mul)*
mul = unary ("*" unary | "/" unary)*
unary      = ("+" | "-")? primary
primary    = num | "("expr ")""
```

Equality stands for == and ! =, and relational stands for <, <=, >, and >=. These non-terminal symbols can be directly mapped to functions using the left join operator parsing pattern.

In the above grammar, we have separated expr and equality to show that the whole expression is equality; we could have written the right side of equality directly on the right side of expr, but I think the above grammar is probably easier to read.

**Column: Simple and verbose code vs. advanced and concise code**

Since recursive descent parsing involves writing code that corresponds almost directly to the generation rules, functions that parse similar rules will have a similar look and feel. The functions we have written so far, relational, equality, add, and mul, should have the same look and feel.

It is probably natural to wonder if the patterns common to such functions can be successfully abstracted using meta-programming techniques such as C macros, C++ templates, higher-order functions, and code generation. In fact, it is possible to do such a thing. However, in this book, we do not dare to do so. The reasons for this are as follows.

Simple code is easy to understand, even if it is somewhat verbose. Even if you end up making similar changes to similar functions later, it's really not much of a hassle. Highly abstracted code, on the other hand, tends to be esoteric because you need to understand the abstraction mechanism first, and then understand how you are using it. For example, if I had started the explanation of this book by writing a function that uses metaprogramming to generate a recursive descent parsing function, the book would have been much more difficult.

You don't always have to aim to write elaborate and concise code. When you aim for that, you tend to make the code difficult to the point where it cannot be made any more difficult.

However, most readers of code do not share the same sense as the author, nor do they need to be so proficient in the first place. However, most readers of code don't share the same sense as the author, and don't need to be that proficient in the first place. It is one of the most important techniques to create a program that is easy to understand and maintain, to dare to write "simple code that could be written in a better way" when necessary.

## Assembly code generation

On x86-64, the comparison is done using the cmp instruction. The code to pop two integers from the stack, perform the comparison, and set RAX to 1 if they are identical and 0 otherwise is as follows

```
pop rdi
pop rax
cmp rax, rdi
sete al
movzb rax, al
```

This code is a short assembly but somewhat lively, so let's go through the code step by step.

The first two lines pop values off the stack, and the third line compares the values that were popped. Where do the comparison results go? In x86-64, the result of the compare instruction is set in a special "flag register". The flag register is updated each time an integer or comparison operation instruction is executed, and contains bits such as whether the result is zero, whether an overflow occurred, and whether the result is less than zero.

The flag register is not a normal integer register, so if you want to set the comparison result in RAX, you need to copy a specific bit of the flag register to RAX. The `sete` instruction sets the specified register (in this case, AL) to 1 if the values of the two registers checked by the previous `cmp` instruction are the same. Otherwise, it sets 0.

AL is a new register name that has not been introduced so far in this book, but in fact AL is just another name for the lower 8 bits of RAX. In fact, AL is just an alias register for the lower 8 bits of RAX, so when `sete` sets a value in AL, RAX is automatically updated. However, when RAX is updated via AL, the upper 56 bits remain at their original value, so if you want to set the entire RAX to 0 or 1, you need to clear the upper 56 bits to zero. It would be better if the `sete` instruction could write directly to RAX, but `sete` is designed to take only 8-bit registers as arguments, so the comparison instruction uses two instructions to set the value in RAX.

Other comparison operators can be implemented by using different instructions instead of `sete`. `Use setl` for `<`, setle `for` <=, `and setne for ! = should` use `setne`.

`> and` >= do not need to be supported by the code generator. Use the parser to swap the two sides and read them as `< and` >=.

### Reference implementation

- [6ddba4be5f633886]

**Column: Flag Registers and Hardware**

This x86-64 specification, where the result of a value comparison is implicitly stored in a special register different from ordinary integer registers, may seem confusing at first. In fact, some RISC processors have an instruction set that sets the result of a value comparison to an ordinary register, rather than having a flag register. RISC-V, for example, has such an instruction set.

However, from the standpoint of implementing hardware, it is very easy to create a flag register for a naive implementation. In other words, when you perform an integer operation, you can branch off the result wires to another logic to see if the result is zero (all lines are 0) or negative (the most significant bit line is 1), and then set the result to each bit of the flag register. That's it. CPUs with flag registers are implemented in just such a way that every time an integer operation is performed, the flag register is updated along with it.

In such a system, the flag register is updated not only by `cmp`, but also by `add` and `sub`. In fact, the entity of `cmp is a` special `sub` instruction that updates only the flag register. `sub rax, rdi, and` then look at the flag register, you can see the size relationship between RAX and RDI, but that would update RAX, so cmp is prepared as a sub instruction that does not write to the integer register. However, this would update RAX, so `cmp is` prepared as a `sub that` does not write to integer registers.

In hardware, however, there is no time penalty for branching a line and using an extra transistor, so the cost of updating the flag register every time is non-existent in a naive hardware implementation. In hardware, there is no time penalty for branching lines and using extra transistors.

# Separate compilation and linking

Up to this stage, we have been developing with only one C file and one shell script for testing. It is not that there is anything wrong with this structure, but the source is getting longer and longer, so let's split it into multiple C files for better visibility. In this step, we will split a single file called 9cc.c into the following five files.

- `9cc.h`: Header files
- `main.c`: `main` function
- `parse.c`: Parser
- `codegen.c`: Code generator

The `main` function is small and could have been put in another C file, but since it does not semantically belong to either `parse.c` or `codegen.c, we will separate it` into separate files.

In this chapter, we will explain the concept of split compilation and its significance, followed by a description of the specific procedure.

What is split compilation?

## Split compilation and its necessity

Split compilation is the process of writing a single program in multiple source files and compiling them separately. In a split compilation, the compiler will read the program fragments and output the corresponding fragments instead of the whole program. A file containing a fragment of a program that is not executable by itself is called an "object file" (extension: `.o). In a` split compilation, the object files are finally stitched together to form a single file. The program that compiles the object files together into a single executable file is called the "linker.

It is important to understand why we need to do split compilation. Actually, technically, there is no such thing as a necessity to split the source. If you give the compiler the entire source code at once, it is logically possible for the compiler to output a complete executable without the help of a linker.

However, in such a case, the compiler really needs to know all the code that the program is using. For example, standard library functions such as `printf are usually` functions written in C by the authors of the standard library, and in order to eliminate the linking step, the source code of such functions must also be given to the compiler input each time. In many cases, compiling the same function over and over again is just a waste of time. Therefore, standard libraries are usually distributed in the form of pre-compiled object files, so that you don't have to recompile them every time. In other words, even if your program consists of a single source code, as long as you are using the standard library, you are actually using split compilation.

If you don't do split compilation, you will have to recompile the entire code even if you change just one line. For a code that is tens of thousands of lines long, compilation can take several tens of seconds. In a large project, there may be more than 10 million lines of source code, so if you compile it as one unit, you will not be able to finish it in a day. Memory is also required in units of 100GiB. Such a build procedure is unrealistic.

Another problem is that simply writing all the functions and variables together in one file is difficult for humans to manage.

It is for the above reasons that split compilation is needed.

### Column: The History of Linka

In 1947, John Mauchly (project leader of ENIAC, the first digital computer) described a program that relocates subprograms read from tape and combines them into a single program.2 The linker's ability to connect multiple fragmented machine language routines into a single program has been needed since the dawn of computers. In 1947, John Mauchly (project leader of ENIAC, the first digital computer) described a program that relocates subprograms read from tape and combines them into a single program.[2]

In 1947, when assembler was not yet in use and people were writing code directly in machine language, linkers were the programs that programmers wanted to create before assembler. In 1947, when assembler was not yet in use and people were writing code directly in machine language, the linker was actually a program that programmers wanted to create before assembler.

## Necessity of header files and their contents

In split compilation, the compiler will only look at one part of the program's code, but this does not mean that the compiler can compile any small fragment of the program. For example, consider the following code.

```
void print_bar(struct Foo *obj) {
 printf("%d\n", obj->bar);
}
```

In the above code, if you know the type of the structure Foo, you can output the corresponding assembly for this code, otherwise you cannot compile this function.

When doing a split compilation, it is necessary to have enough information in each C file to be able to compile the individual files. However, if you write all the code in different files, it will not be a split compilation, so you need to select the information to some extent.

As an example, let's consider what information needs to be included in order to output code that calls a function in another C file. The compiler needs the following information

- In the first place, we need information that an identifier is the name of a function.
- The function call code output by the compiler sets the arguments to the registers in a certain order and then jumps to the beginning of another function using the call instruction. Depending on the argument type, it may also convert an integer to a floating point number. If the type or number of arguments is wrong, an error message should also be printed. Therefore, the number of arguments to the function and the type of each argument are required.
- The code of the calling function is not required when compiling the calling function, because whatever is done beyond the calling function will simply return in time for the caller.
- The address to jump to with `call is` not known at the time of split compilation, but the assembler outputs a call instruction that jumps to address 0 for now, and leaves information in the object file that says, "Modify the Xth byte of the object file with the address of the function named Y. The linker can see this information and use it to modify the object file. The linker looks at that information, decides the layout of the executable file, and then performs binary patching of the program fragment to modify the jump destination address (this operation is called "relocating"). Therefore, to split-compile, the name of the function is needed, but the address of the function is not.

Summarizing the above requirements, we can say that as long as we omit the `{ ... } in the function body, we have enough` information to call the function.

Such an omission of the function body is called a "declaration" of the function. A declaration only tells the compiler the type and name, but does not contain the code of the function. For example, the following is a declaration of strncmp.

```
int strncmp(const char *s1, const char *s2, size_t n);
```

The compiler can know the existence of strncmp and its type by looking at the above line. In contrast to declarations, those containing function code are called "definitions".

Function declarations must be accompanied by the keyword extern to indicate the declaration, and

```
extern int strncmp(const char *s1, const char *s2, size_t n);
```

However, in the case of functions, the function body is omitted to distinguish between declarations and definitions, so the extern may be omitted.

Since the type of the argument is all that is needed, the name can be omitted in the declaration, but it is common to write the name in the declaration as well to make it easier for people to understand.

As another example, let's consider the type of structure. If you have two or more C files that use the same struct, you need to write the same struct declaration in each C file. if the struct is only used in one C file, the other C files don't need to know about its existence.

In C, the declarations that are needed to compile other C files are written in a header file (.h extension). The #include line will be replaced with the contents of the foo.h file.

Typedefs are also used to tell the compiler the type information. These also need to be written in the header file if they are used in multiple C files.

The compiler does not output any particular assembly when it reads a declaration. This is because a declaration is information necessary to use a function or variable contained in another file, and does not itself define a function or variable.

Given what we've discussed so far about split compilation, you can see what all this talk about "when you use printf, you should write #include <stdio.h> as a spell" is actually doing: the C standard library is passed implicitly to the linker, so The linker can link an object file containing printf function calls to create an executable file. On the other hand, the compiler does not know anything about `printf by default`, because `printf` is not a built-in function, and there is no specification that the standard library header file is automatically loaded, so the compiler knows nothing about `printf` right after it starts. The compiler does not know anything about printf. By including the header file that comes with the C standard library, the compiler knows about the existence of `printf` and its type, and can compile `printf` function calls.

**Column: One-pass compilers and forward declarations**

The C language specification allows the compiler to compile functions one by one, starting from the beginning, instead of reading the whole file. Therefore, every function must be able to be compiled with only the information written up to the point where it appears in the file. Therefore, if you want to use a function defined behind a file, you need to write the declaration of the function beforehand. Such a declaration is called a ″forward declaration".

By devising the order in which functions are written in the file, it is possible to get away with writing most forward declarations, but if you want to write functions that are mutually recursive, forward declarations are required.

The C language's specification to allow compiling without reading the entire file made sense in the days when main memory was very small, but it is now obsolete. If the compiler was a little smarter, it would be able to avoid writing declarations for definitions that are written in the same file. However, this behavior is part of the language specification, so you need to keep it in mind.

## link error

When the object file is finally put together and passed to the linker, it must contain enough information to make up the program as a whole.

If the program contains only the declaration of the function `foo`, but no definition, then the individual C files can be compiled normally, including the code that calls `foo`. However, when the linker finally tries to create a complete program, it will produce an error because the part of the program that should be modified with the address of `foo` `cannot be` modified because there is no `foo`.

An error when linking is called a link error.

If multiple object files contain the same function or variable, it will also result in a link error. This is because the linker is not sure which one to choose when there is a duplicate. This kind of duplication error often occurs when the wrong definition is written in the header file. This is because the header file is included in multiple C files, and if there is a definition in the header file, it is the same as if the definition is duplicated in multiple C files. To eliminate such errors, please write only declarations in the header file, and move the entities to any one C file.

**Column: Duplicate definitions and link errors**

When there are duplicate definitions, it is possible for the linker to choose one of the definitions and ignore the rest. In such a linker, duplicate definitions will not result in an error.

In the actual object file, it is possible to choose whether to allow duplication or not for each definition, and inline functions and C++ template expansion results can be included in the object file with duplication allowed. The formatting of object files and the behavior of the linker is surprisingly complex, with many exceptions, but those are the exceptions. By default, it is normal for duplicate definitions to result in an error.

## Declaring and defining global variables

Since our compiler does not yet have global variables, we do not yet have an assembly example for global variables, but global variables are almost the same as functions at the assembly level. However, global variables are almost the same as functions at the assembly level. Therefore, just like functions, global variables have a distinction between definitions and declarations. If the body of a variable is duplicated in multiple C files, it usually results in a linking error.

Global variables are assigned to non-executable memory areas by default, so jumping to them will cause the program to crash with a segmentation fault, but essentially there is no other difference between data and code. You can read a function as data at runtime, like a global variable, or you can execute data as code by changing the memory attributes to allow execution and jumping to the data.

Let's see in real code that both functions and global variables are essentially just data that reside in memory. In the following code, an identifier called `main` is defined as a global variable, and the contents of `main` are in x86-64 machine language.

```
char main[] = "\x48\xc7\xc0\x2a\x00\x00\x00\xc3";
```

Let's save the above C code as a file named `foo.c`, compile it, and check its contents using `objdump`. `objdump` by default only prints the contents of global variables in hexadecimal, but by passing the `-D` option, we can force the data to be disassembled as code. By passing the -D option, we can force the data to be disassembled as code.

```
$ cc -c foo.c
$ objdump -D -M intel foo.o
Disassembly of section .data:

0000000000000000 <main>:
    0:    48 c7 c0 2a 00 00 00     mov     rax,0x2a
    7:    c3                       ret
```

The default behavior of data being mapped to the execution prohibited area can be changed by passing the options `-Wl,--omagic` at compile time. Let's try to generate an executable file using this option.

```
$ cc -static -Wl,--omagic -o foo foo.o
```

Since both functions and variables are just labels in assembly and belong to the same namespace, the linker does not care which one is a function and which one is data when putting together multiple object files. Therefore, even if `main` is defined as data at the C level, the link will succeed just as if `main` were a function.

Let's run the generated file.

```
$ . /foo
$ echo $?
42
```

As you can see above, the value of 42 is correctly returned, which means that the content of the global variable called `main` has been executed as code.

In C syntax, the extern is a declaration for a global variable. The following is a declaration of the global variable foo of type int.

```
extern int foo;
```

If you write a program that contains foo, you will have to write the above lines in a header file. Then, you will define foo in one of your C files. The following is the definition of foo.

```
int foo;
```

Note that in C, global variables that are not given an initialization expression are supposed to be initialized with 0, so such variables can be initialized with 0, {0, 0, ...} ...}, "\0\0\0\0..." etc. It is semantically the same as being initialized with

When writing initialization expressions, such as int foo = 3, please write the initialization expression in the definition only. The declaration is to tell the compiler only the type of the variable, so there is no need for a specific initialization expression. The compiler does not output any particular assembly when it sees the declaration of a global variable, so it does not need to know how its contents are initialized on the fly.

If the initialization expression is omitted, the declaration and definition of a global variable will look similar because the only difference is the presence or absence of extern, but declaration and definition are different things. However, declaration and definition are two different things.

### Column: F00F bug in Intel CPUs

Prior to 1997, Intel Pentium processors had a serious bug that caused the CPU to hang completely when executing a 4-byte instruction named F0 0F C7 C8.

There is no official assembly instruction corresponding to this 4-byte instruction, but if we dare to write it down as an assembly instruction, it will be called lock cmpxchg8b eax. 0F C7 C8 is an instruction called cmpxchg8b eax, which is an instruction to exchange an 8-byte value between registers and memory atomically (in a way that is unobservable to other cores even in multi-core). F0 is an additional information called lock prefix, which has the effect of making the immediately following instruction atomic. However, since cmpxchg8b is atomic in nature, lock cmpxchg8b eax is a redundant and illegal way of writing instructions. Therefore, such assembly instructions are supposed to be grammatically nonexistent, and the byte sequence `F0 0F C7 C8` never appears in a normal program, and Intel could not notice this bug before mass production of the processor.

Using the hack of writing the main function as data, the code to reproduce the F00F bug can be written in C in the following one line.

```
char main[] = "\xf0\x0f\xc7\xc8";
```

In modern x86, this function is harmless, but on a Pentium back in 1997, anyone could easily hang up the entire system with this one line program.

If the PC is completely occupied by an individual, the F00F bug is not a big problem, but if the CPU is being used in a shared manner, such as in what is now called a cloud, this bug is fatal. Initially, it was thought that the F00F bug could not be fixed and that the only solution was to recall and replace the CPU, but later a tricky method was developed at the exception handler level of the OS kernel to work around the bug, and fortunately for Intel, product replacement was avoided.

C standard library and archive files

Step 8: Split the files and modify the Makefile

## Splitting a file

Try to split the files in the configuration shown at the beginning of this chapter. `9cc.h is the` header file. Depending on the structure of your program, you may have one `.h file` for each `.c` file, but the extra declarations won't do any harm, so there is no need to manage dependencies in such detail here. file and include it in all C files as `#include "9cc.h"`.

## Change the Makefile

Now that we have changed the program to multiple files, let's also update the `Makefile`. `The Makefile` below is for compiling and linking all the .c files in the current directory to create an executable called 9cc. It assumes that there is only one header file for the project, 9cc.h, and that the header file is included in all the .c files.

```
CFLAGS=-std=-c11 -g -static
SRCS=$(wildcard *.c)
OBJS=$(SRCS:.c=.o)

9cc: $(OBJS)
 $(CC) -o 9cc $(OBJS) $(LDFLAGS)

$(OBJS): 9cc.h

test: 9cc
        . /test.sh

Clean:
        rm -f 9cc *.o *~ tmp*

.PHONY: test clean
```

Note that the indentation in the Makefile must be a tab character.

Although make is a sophisticated tool that does not necessarily need to be mastered, it is useful to be able to read the above Makefile in various situations. In this section, we will explain the above Makefile.

In a Makefile, lines separated by colons and zero or more lines of commands indented by tabs constitute a single rule. The name before the colon is called the "target". The zero or more file names after the colon are called dependent files.

If you run "make foo", make will try to create a file named foo. If the specified target file already exists, make will re-execute the target rules only if the target file is older than the dependent file. This achieves behaviors such as regenerating binaries only when the source code has changed.

. PHONY is a special name for a dummy target. make test and make clean are not run to create files like test and clean, but usually make doesn't know that, so if a file with a name like test or clean happens to exist, make test and make clean will do nothing. If a file with a name like test or clean happens to exist, make test or make clean will not do anything. PHONY, you can tell make that you don't really want to create a file with that name, and that it should execute the rule command regardless of whether the target file exists or not.

CFLAGS, SRCS, and OBJS are variables.

CFLAGS is a variable that is recognized by the built-in rules of make, and you can write command line options to pass to the C compiler. Here we are passing the following flags.

- -std=c11: Tells you that the source code is written in C11, the latest C standard.
- -g: Output debug information
- -static: static linking

The "wildcard" used in the right side of SRCS is a function provided by make, which expands to a filename matching the function argument. $(wildcard *. c) will be expanded to main.c parse.c codegen.c at the moment.

The right side of OBJS uses a variable substitution rule, which generates a value that replaces .c in SRC with .o. SRCS is main.c parse.c codegen.c, so OBJS is main.o parse.o codegen.o.

With all this in mind, let's trace what happens when we run make 9cc. make tries to generate the target specified as an argument, so creating a 9cc file is the ultimate goal of the command (if there is no argument, the first rule is chosen, so 9cc does not have to be specified). Make will therefore traverse the dependencies and try to build the missing or outdated files.

The 9cc dependency file is the .o file that corresponds to the .c file in the current directory. If there is still a .o file from the last time make was run, and it has a newer timestamp than the corresponding .c file, make will not bother to re-run the same command. Only if the .o file does not exist or the .c file is newer, will it run the compiler to generate the .o file.

The rule $(OBJS): 9cc.h means that all .o files depend on 9cc.h. Therefore, any change to 9cc.h will cause all .o files to be recompiled.

**Column: Various meanings of static keywords**

The static keyword in C is mainly used for two purposes: 1.

1. Attach static to local variables so that their values are preserved even after exiting the function.
2. Attaching static to a global variable or function to make the scope of that variable or function a file scope

This is one of the points of confusion when learning C because the same keywords are used for these two usages even though there is nothing in common between them. Ideally, we should have used different keywords such as persistent for use 1 and private for use 2. Ideally, it would have been better to use private as the default for usage 2, and use public for variables and functions with a global scope.

The reason for C's use of keywords is compatibility with previously written code assets; adding new keywords to the language, such as private, would prevent existing programs that use those keywords as variable or function names from compiling. Instead of adding more keywords, C decided to reuse existing keywords in different contexts.

If I had made the decision to add new keywords instead of using static keywords at some point in the 1970s, I would not have had to change a great deal of code, but it is a difficult question to answer when I think about what I would have done.

# Functions and local variables

In this chapter, we will implement functions and local variables. It also implements a simple control structure. At the end of this chapter, you should be able to compile code like the following

```
// Add m to n
sum(m, n) {
 acc = 0;
 for (i = m; i <= n; i = i + 1)
     acc = acc + i;
 return acc;
}

main() {
 return sum(1, 10); // return 55
}
```

The above code still has a gap from C, but I think we can say that it is still pretty close to C.

### Step 9: Local variable with one character

In the previous section, we were able to create a compiler for a language that can do four arithmetic operations. In this section, we will add a feature to that language so that we can use variables. Specifically, our goal is to be able to compile multiple statements that contain variables, as shown below.

```
a = 3;
b = 5 * 6 - 8;
a + b / 2;
```

We will use the result of the last expression as the calculation result for the entire program. I would say that the language of this is starting to look a lot more like a ″real language″ than a language with only four arithmetic operations.

In this chapter, we will first explain how to implement variables, and then we will implement variables incrementally.

## Variable area on the stack

Variables in C reside in memory. You can say that variables are named after memory addresses. By naming the memory address, we can say ″access variable a" instead of "access memory address 0x6080".

However, the local variables of a function must exist separately for each function call. For implementation convenience alone, it may be easy to set the address as ″the local variable a of function f is placed at 0x6080", but this will not work well when f is called recursively. In order to have a separate local variable for each function call, in C, local variables are supposed to be placed on the stack.

Let's consider the contents of the stack with a concrete example. Suppose we have a function f with local variables a and b, and some other function calls f. The call instruction of the function call puts the return address on the stack, so the top of the stack at the time f is called will contain the return address. In addition to that, it is assumed that the stack originally contains some other value. We'll use ″...……″, since the specific value is not important here. The diagram looks like this

| | |
|---|---|
| … | |
| return address | ← RSP |

We will use the notation "← RSP" to indicate that the current RSP register value points to this address. `a` and `b are` each 8 bytes in size.

The stack grows downward. In order to allocate space for `a` and `b` from this state, we need to push down the RSP by two variables, or 16 bytes in total. Doing so will look like this

| | |
|---|---|
| … | |
| return address | |
| A | |
| b | ← RSP |

With the above layout, the value of RSP+8 would allow access to `a`, and the value of RSP would allow access to `b`, `and so` on. The memory area allocated for each function call is called a "function frame" or "activation record".

How many bytes to change in the RSP and in what order to place variables in the allocated area are not visible to other functions, so you can decide appropriately according to the compiler's implementation convenience.

Basically, a local variable is implemented as something as simple as this.

However, this method has one drawback, and we will use one more register for the actual implementation. Recall that in our compiler (and others like it), the RSP may change while the function is being executed. 9cc pushes/pops the result of the calculation in the middle of the expression onto the stack using the RSP, so the value of the RSP changes frequently. Therefore, `a` and `b` `cannot` be accessed at a fixed offset from the RSP.

A common way to solve this is to prepare a register, separate from the RSP, that always points to the start of the current function frame. In x86-64, it is customary to use the RBP register as the base register.

The base pointer should not change during the execution of a function (that is the reason for having a base pointer). You cannot call another function from a function and come back with a different value, so you need to save the original base pointer for each function call and write it back before returning.

The following figure shows the state of the stack in a function call using a base pointer. Let's assume that a function `g with` local variables `x` and `y` calls `f`. While `g is` executing, the stack looks like this

| | |
|---|---|
| … | |
| Return address of g | |
| RBP at the time of invocation of g | ← RBP |
| x | |
| Y | ← RSP |

Calling `f` from here will result in the following state.

| |
|---|
| … |
| Return address of g |
| RBP at the time of invocation of g |
| x |

| | |
|---|---|
| Y | |
| Return address of f | |
| RBP at the time of invocation of f | ← RBP |
| A | |
| b | ← RSP |

In this way, we can always access `a` with the address RBP-8 and `b` with the address RBP-16. Concretely, if we consider an assembly that creates such a stack state, the compiler should output the following assembly at the beginning of each function.

```
push rbp
mov rbp, rsp
sub rsp, 16
```

This kind of fixed instruction that the compiler outputs at the beginning of a function is called a ″prologue". The number 16 must be set to a value that matches the number and size of variables in each function.

The state of the stack for each instruction is shown below.

1. Stack immediately after calling `f` with `call`

| | |
|---|---|
| ... | |
| Return address of g | |
| RBP at the time of invocation of g | ← RBP |

| | |
|---|---|
| x | |
| Y | |
| Return address of f | ← RSP |

2. Stack after executing `push rbp`

| | |
|---|---|
| … | |
| Return address of g | |
| RBP at the time of invocation of g | ← RBP |
| x | |
| Y | |
| Return address of f | |
| RBP at the time of invocation of f | ← RSP |

3. Stack after executing `mov rbp, rsp`

| | |
|---|---|
| … | |
| Return address of g | |

| | |
|---|---|
| RBP at the time of invocation of g | |
| x | |
| Y | |
| Return address of f | |
| RBP at the time of invocation of f | ← RSP, RBP |

4. Stack after executing `sub rsp, 16`

| | |
|---|---|
| ... | |
| Return address of g | |
| RBP at the time of invocation of g | |
| x | |
| Y | |
| Return address of f | |
| RBP at the time of invocation of f | ← RBP |
| A | |

| | |
|---|---|
| b | ← RSP |

When returning from a function, write the original value back to the RBP so that the RSP is pointing to the return address, and then call the `ret` instruction (the `ret` instruction is an instruction that pops an address from the stack and jumps to it). The code to do that can be written concisely as follows.

```
mov rsp, rbp
pop rbp
ret
```

Such a fixed instruction that the compiler outputs at the end of a function is called an "epilogue".

The status of the stack while executing the epilogue is shown below; the stack area below the address pointed to by the RSP is omitted in the figure since it can be regarded as invalid data.

1. Stack before executing `mov rsp, rbp`

| | |
|---|---|
| … | |
| Return address of g | |
| RBP at the time of invocation of g | |
| x | |
| Y | |
| Return address of f | |
| RBP at the time of invocation of f | ← RBP |

| | |
|---|---|
| A | |
| b | ← RSP |

2. Stack after executing `mov rsp, rbp`

| | |
|---|---|
| … | |
| Return address of g | |
| RBP at the time of invocation of g | |
| x | |
| Y | |
| Return address of f | |
| RBP at the time of invocation of f | ← RSP, RBP |

3. Stack after running `pop rbp`

| | |
|---|---|
| … | |
| Return address of g | |
| RBP at the time of invocation of g | ← RBP |

| | |
|---|---|
| x | |
| Y | |
| Return address of f | ← RSP |

4. Stack after executing `ret`

| | |
|---|---|
| … | |
| Return address of g | |
| RBP at the time of invocation of g | ← RBP |
| x | |
| Y | ← RSP |

Thus, executing the epilogue restores the state of the stack of the calling function `g`. The call instruction piles the address of the next instruction after the `call` instruction itself on the stack. The call instruction pops the address of the next instruction after the call itself on the stack, and the epilogue `ret` pops that address and jumps to it, so that the execution of function `g` resumes from the next instruction after the `call`. This kind of behavior is completely consistent with the behavior of functions we know.

This is how function calls and function local variables are realized.

### Column: Direction of stack extension

The stack in x86-64 grows from the larger address to the smaller address, as explained above. It seems more natural to grow in the opposite direction, i.e., up the stack, but why is the stack designed to grow down?

Actually, there is no technical necessity for the stack to grow downward. In most actual CPUs and ABIs, the stack starts at the upper address and grows downward, but there are some architectures, albeit very minor, where the stack grows in the opposite direction. For example, in 8051 microcontrollers, PA-RISC [ABI3], [Multics4], etc., the stack grows in the upper address direction.

However, the design of the stack growing downward is not particularly unnatural.

When the CPU starts executing a program from a clean slate immediately after power-on, the address at which it starts execution is usually determined by the CPU specifications. In a typical design, the CPU starts execution from a lower address, such as address 0. This usually means that the program code is put together and placed at the lower address. If we place the two as far apart as possible so that the stack does not grow and cover the program code, we would design the stack to be placed at the upper address and grow towards the center of the address space. In this way, the stack will grow downward.

Of course, we can again consider a different design than the CPU above, and then the more natural arrangement would be to grow the stack upwards. To be honest, it doesn't really matter which way you look at it, it's just that the general perception in the industry is that the machine stack grows downward.

## Change the tokenizer.

Now that we know how to implement variables, let's start implementing them. However, supporting an arbitrary number of variables would suddenly become too difficult, so we will limit the variables in this step to a single lowercase letter, and assume that all variables will always exist: variable `a` is RBP-8, variable `b` is RBP-16, variable `c` is RBP-24, and so on. Since there are 26 letters in the alphabet, if we decide to push down the RSP for 26 x 8, or 208 bytes, when we call the function, we will have enough space for all the single letter variables.

Now let's get started with the implementation. First, we need to modify the tokenizer so that we can tokenize a single character variable in addition to the existing grammar elements. To do so, we need to add a new token type. Since the name of the variable can be read from the `str` member, there is no need to specifically add a new member to the `Token` type. As a result, the token type will look like this

```
enum {
  TK_RESERVED, // symbol
  TK_IDENT,    // identifier
  TK_NUM,    //integer token
  TK_EOF,    // token representing the end of the input
} TokenKind;
```

Make a change to the tokenizer so that it will create a token of type TK_IDENT if it is a lower case letter. You should be able to add the following if statement to the tokenizer

```
if ('a' <= *p && *p <= 'z') {
  cur = new_token(TK_IDENT, cur, p++);
  cur-> len = 1;
  Continue;
}
```

## Changing the parser

In recursive descent parsing, if we knew the grammar, we could mechanically map it to a function call. Therefore, in order to think about the changes we should make to the parser, we need to consider what the new grammar looks like with the addition of variable names (identifiers).

Let's say the identifier is ident. This is a terminator, just like num. Variables can be used anywhere numbers can be used, so if we change the name from num to num | ident, we can use variables in the same place as numbers.

In addition to that, we need to add substitution expressions to the grammar. Variables are no use if you can't assign them, so you want to have a grammar that allows expressions like a=1. In this case, let's make a grammar that can be written like a=b=1 to match C.

Furthermore, we want to be able to write multiple statements separated by semicolons, so the resulting new grammar will look like this

```
program    = stmt*
stmt = expr ";"
expr       = assign
assign     = equality ("=" assign)?
equality   = relational ("==" relational | "! =" relational)*
relational = add ("<" add | "<=" add | ">" add | ">=" add)*
add        = mul ("+" mul | "-" mul)*
mul = unary ("*" unary | "/" unary)*
unary      = ("+" | "-")? primary
```

```
primary    = num | ident | "("expr ")""
```

First, make sure that your programs such as `42;` and `a=b=2; a+b; conform to` this grammar. After that, modify the parser you have created so far so that it can parse the above grammar. At this stage, it will be able to parse expressions like `a+1=5, which is the` correct behavior. The elimination of such semantically incorrect expressions will be done in the next pass. As for modifying the parser, there is nothing tricky, and you should be able to do the same thing as before, just map the grammar elements directly to function calls.

Now that we have multiple expressions separated by semicolons, we need to store multiple nodes as the result of parsing somewhere. For now, you can prepare the following global array and store the nodes of the parsing result there in order. The last node should be filled with NULL so that we know where the end is. A part of the new code to be added is shown below.

```
Node *code[100];

Node *assign() {
 Node *node = equality();
 if (consume("="))
     node = new_node(ND_ASSIGN, node, assign());
 return node;
}

Node *expr() {
 return assign();
}

Node *stmt() {
 Node *node = expr();
 expect(";");
 return node;
}

void program() {
 int i = 0;
 while (! at_eof())
     code[i++] = stmt();
 code[i] = NULL;
}
```

We need to be able to represent a new ″node representing a local variable″ in the abstract syntax tree. To do so, let's add a new type of local variable and a new member of the node. For example, it should look like this In this data structure, the parser will create and return a node of type `ND_LVAR` for an identifier token.

```
typedef enum {
 ND_ADD,    // +
 ND_SUB,    // -
 ND_MUL,    // *
 ND_DIV,    // /
 ND_ASSIGN, // =
 ND_LVAR,    // local variable
 ND_NUM,    // integer
} NodeKind;

typedef struct Node Node;

// Node of an abstract syntax tree
struct Node {
 NodeKind kind; // type of the node
 Node *lhs;    // left side
 Node *rhs;    // right side
 int val;    // use only if kind is ND_NUM
 int offset;    // only used if kind is ND_LVAR
};
```

`offset` is a member that represents the offset of the local variable from the base pointer. For now, variable `a` is RBP-8, `b` is RBP-16……, and so on. Since local variables are in fixed positions determined by their names, the offset can be determined at the parsing stage. Here is the code that reads an identifier and returns a node of type `ND_LVAR`.

```
Node *primary() {
 ...

 Token *tok = consume_ident();
 if (tok) {
     Node *node = calloc(1, sizeof(Node));
     node->kind = ND_LVAR;
     node->offset = (tok-> str[0] - 'a' + 1) * 8;
 return node;
 }

 ...
```

### Column: ASCII Code

In ASCII code, characters are assigned to numbers from 0 to 127, and a table of character assignments in ASCII code is shown below.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | NUL | SOH | STX | ETX | EOT | ENQ | ACK | BEL |
| 8 | BS | HT | NL | VT | NP | CR | SO | SI |
| 16 | DLE | DC1 | DC2 | DC3 | DC4 | negative acknowledgement | SYN | ETB |
| 24 | CAN | EM | SUB | ESC | FS | GS | RS | US |
| 32 | sp | ! | " | # | $ | percent | &. | ' |
| 40 | ( | ) | * | + | . | -Mr. | . | / |
| 48 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 56 | 8 | 9 | . | . | < | = | > | ? |
| 64 | @ | A | B | C | D | E | F | G |
| 72 | H | I | J | K | L | M | N | O |
| 80 | P | Q | R | S | T | U | V | W |
| 88 | X | Y | Z | [ | sea bream (Sparidae) | ] | ^ | _ |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 96 | ` | A | b | c | d | e | f | g |
| 104 | H | I | J | K | l | m | n | O |
| 112 | p | q | r | s | t | u | V | W |
| 120 | x | Y | z | { | \| | } | ~ | DEL |

The characters in 0 to 31 are control characters. Nowadays, with the exception of NUL and newline characters, these control characters are rarely used, and most of them are just wasting space in the character code, but back in 1963, when the ASCII code was developed, these control characters were actually in common use. When the ASCII standard was being developed, there was even a proposal to include more control characters instead of the lowercase alphabet.[5]

Numbers are assigned to 48-58, uppercase letters to 65-90, and lowercase letters to 97-122. Notice that these characters are assigned to consecutive codes. In other words, 0123456789, abcdefg... are contiguous in the character code. It seems natural to put characters with a defined order in such consecutive positions, but in the major character codes of the time, such as EBCDIC, alphabets were not consecutive in the code due to punch cards.

In C, a character is just a small value of an integer type, which is no different in meaning than writing the code corresponding to a character as a number. So, assuming ASCII, for example, `'a'` is equivalent to 97 and `'0'` is equivalent to 48. In the above code, there was an expression where `a` was subtracted from a character as a number, and in that way, we can calculate how many characters away from a a a given character is. This is a technique that is possible because the alphabet is arranged consecutively in the ASCII code.

## Left-sided and right-sided values

Since assignment expressions, unlike other binary operators, require special handling of the left-hand side value, let's discuss that here.

The left-hand side of an assignment expression does not mean that any expression is allowed. Assignments such as a=2 are allowed, but statements such as (a+1)=2 are illegal. 9cc does not yet have pointers or structures, but if it did, assignments to the destination of a pointer, such as *p=2, or to a member of a structure, such as a. If they do exist, assignments to the destination pointed to by pointers, such as *p=2, and assignments to members of structures, such as b=2, must be allowed as legitimate. How can we distinguish between such legitimate and illegitimate expressions?

There is a simple answer to that question: in C, the only thing that can come on the left side of an assignment expression is basically an expression that specifies the address of memory.

Variables can be written on the left-hand side of assignments, since variables exist in memory and have addresses. Similarly, a pointer reference such as *p can also be written on the left-hand side, since the value of p is an address. a.b is a member access of a structure, which points to a memory address that is the offset of the member b from the starting position of the structure a in memory. It can be written on the left side.

On the other hand, the result of an expression such as a+1 is not a variable, which means that it cannot be used as an expression to address memory. These temporary values may actually exist only in registers and not in memory, and even if they do exist in memory, they cannot usually be accessed with a fixed offset from a known variable. For these reasons, writing, for example, &(a+1) does not allow you to get the address of the result of a+1, and will result in a compilation error. Such an expression cannot be written on the left side of an assignment statement.

A value that can be written on the left side is called a left value, and a value that cannot is called a right value. Left values and right values are sometimes called lvalue and rvalue, respectively. In our current language, only variables are left-hand side values, and all other values are right-hand side values.

When generating code for a variable, you can think of the left side value as the starting point. If a variable appears as the left-hand side of an assignment, we can try to calculate the address of the variable as the left-hand side value, and store the result of the right-hand side evaluation against that address. This allows us to implement an assignment expression. If the variable appears in any other context, convert the left-hand side value to the right-hand side value by calculating the address of the variable in the same way and then loading the value from that address. This will allow you to get the value of the variable.

## How to load a value from an arbitrary address

In the code generation so far, we have only accessed the memory on the top of the stack, but for local variables, we need to access any location on the stack. In this section, we will explain how to access the memory.

The CPU can load and store values from any address in memory, not just the stack top.

To load a value from memory, use the syntax `mov dst, [src]`. This instruction means "consider the value in the src register as an address, load the value from there, and store it in dst. For example, `mov rdi, [rax]` would load the value from the address in RAX and set it in RDI.

To store, use the syntax `mov [dst], src`. This instruction means "consider the value in the dst register as an address, and store the value in the src register there. For example, `mov [rdi], rax` means to store the value of RAX to the address in RDI.

Since `push` and `pop` are instructions that implicitly consider RSP as an address and perform memory access, they can actually be rewritten in multiple instructions using ordinary memory access instructions. So, for example, `pop rax` can be rewritten as

```
mov rax, [rsp].
add rsp, 8
```

and `push rax` is the same as the two instructions

```
sub rsp, 8
mov [rsp], rax
```

This is the same as the two commands

## Change the code generator

Using the knowledge we have gained so far, let's make some changes to the code generator so that it can handle expressions containing variables. In this case, we will add a function that evaluates the expression as the left-hand side value. The function `gen_lval` in the code below does just that. `gen_lval` calculates the address of a variable and pushes it onto the stack if the given node points to a variable. Otherwise, it prints an error. This will eliminate expressions such as `(a+1)=2`.

To use a variable as a right-hand side value, first evaluate it as a left-hand side value, then consider the result of the calculation at the top of the stack as an address, and load the value from that address. The code is shown below.

```
void gen_lval(Node *node) {
 if (node-> kind ! = ND_LVAR)
     error("Left side value of assignment is not a variable");

 printf(" mov rax, rbp\n");
 printf(" sub rax, %d\n", node->offset);
 printf(" push rax\n");
}

void gen(Node *node) {
 switch (node->kind) {
 case ND_NUM:
 printf(" push %d\n", node->val);
 return;
 case ND_LVAR:
 gen_lval(node);
 printf(" pop rax\n");
 printf(" mov rax, [rax]\n");
 printf(" push rax\n");
 return;
 case ND_ASSIGN:
 gen_lval(node-> lhs);
     gen(node-> rhs);

 printf(" pop rdi\n");
 printf(" pop rax\n");
 printf(" mov [rax], rdi\n");
 printf(" push rdi\n");
 return;
 }

 gen(node-> lhs);
 gen(node-> rhs);

 printf(" pop rdi\n");
 printf(" pop rax\n");
 switch (node->kind) {
 case '+':
 printf(" add rax, rdi\n");
 break;
 case '-':
 printf(" sub rax, rdi\n");
 break;
 case '*':
 printf(" imul rax, rdi\n");
 break;
 case '/':
 printf(" cqo\n");
 printf(" idiv rdi\n");
 }

 printf(" push rax\n");
}
```

## Change the main function

Now that all the parts are in place, let's change the `main` function as well and try to get the compiler to work.

```
int main(int argc, char **argv) {
 if (argc != 2) {
     error("The number of arguments is incorrect.");
 return 1;
 }

 // Tokenize and parse.
 // result will be saved in code
 user_input = argv[1];
 tokenize();
 program();

 // Output the first half of the assembly
 printf(".intel_syntax noprefix\n");
 printf(".globl main\n");
 printf("main:\n");

 // Prologue
 // Allocate an area for 26 variables
 printf(" push rbp\n");
 printf(" mov rbp, rsp\n");
 printf(" sub rsp, 208\n");
 // Code generation starting with the first expression
 for (int i = 0; code[i]; i++) {
     gen(code[i]);

 // There is one value left on the stack as a result of evaluating the
expression
 // should be popped so that the stack does not overflow
 printf(" pop rax\n");
 }

 // Epilogue
 // Since the result of the last expression is still in RAX, it will be the
return value.
 printf(" mov rsp, rbp\n");
 printf(" pop rbp\n");
 printf(" ret\n");
 return 0;
}
```

## Step 10: Multi-character local variables

In previous chapters, we decided to use a single character for variable names and treat the 26 local variables from a to z as if they always existed. In this section, we will support identifiers with names longer than one character so that code like the following can be compiled

```
foo = 1;
bar = 2 + 3;
return foo + bar; // return 6
```

Variables are assumed to be available without definition. Therefore, the parser needs to determine for each identifier whether it has seen before, and if it is new, automatically assign the variable to the stack area.

First, change the tokenizer to read identifiers consisting of multiple characters as tokens of type `TK_IDENT`.

We will represent variables as a concatenated list, with a structure called `LVar` representing a single variable, and a pointer called `locals` representing the first element. The code looks like this

```
typedef struct LVar LVar;
```

```
// type of local variable
struct LVar {
 LVar *next; // next variable or NULL
 char *name; // name of the variable
 int len;    // length of the name
 int offset; // offset from RBP
};

// Local variable
LVar *locals;
```

In the parser, when a token of type `TK_IDENT` is encountered, the parser checks if the identifier has ever been encountered before. If the variable has appeared before, it will use the `offset` of that variable. If it is a new variable, we create a new `LVar`, set a new offset, and use that offset.

The function to find variables by name is shown below.

```
// Search for variables by name. If not found, return NULL.
LVar *find_lvar(Token *tok) {
 for (LVar *var = locals; var; var = var->next)
 if (var-> len == tok-> len & & ! memcmp(tok->str, var->name, var->
len))
 return var;
 return NULL;
}
```

In the parser, you should be able to add a code like the following

```
Token *tok = consume_ident();
if (tok) {
 Node *node = calloc(1, sizeof(Node));
 node->kind = ND_LVAR;

 LVar *lvar = find_lvar(tok);
 if (lvar) {
     node->offset = lvar-> offset;
 } else {
 lvar = calloc(1, sizeof(LVar));
 lvar->next = locals;
 lvar->name = tok-> str;
 lvar-> len = tok-> len;
 lvar->offset = locals->offset + 8;
     node->offset = lvar-> offset;
     locals = lvar;
 }
 return node;
}
```

### Column: Frequency of Machine Language Commands

If you look at the assemblies output by 9cc, you will notice that there are many data movement instructions such as `mov` and `push`, and relatively few instructions that perform "real calculations" such as `add` and `mul`. One reason for this is that 9cc does not perform optimization and outputs useless data movement instructions, but in fact, even with an optimizing compiler, data movement instructions are the most commonly output. I disassembled all the executable files in `/bin` in my environment and counted the number of instructions, and the graph below shows the results.

Frequency of occurrence of commands

As you can see, the `mov` instruction alone accounts for a whopping 30% of all instructions. A computer is a data processing machine, and the most frequent part of data processing is data movement. If you consider that "moving data to the right place" is one of the essentials of data processing, this large number of `mov` instructions may seem to be a natural result, but many readers may find it surprising.

## Step 11: Return statement

In this chapter, we will add a `return` statement so that the following code can be compiled.

```
a = 3;
b = 5 * 6 - 8;
return a + b / 2;
```

We will assume that the `return` statement can be written in the middle of the program. As in normal C, the execution of the program will be terminated at the first `return` and return from the function. For example, the following program returns the value of the first `return`, i.e. 5.

```
return 5;
return 8;
```

In order to implement this feature, let's first consider what the syntax will look like with the addition of `return`. In the past, a statement was just an expression, but the new grammar will allow a `return <expression>;`. So the new grammar will look like this

```
program = stmt*
stmt = expr ";"
         | "return" expr ";"
...
```

In order to implement this, the tokenizer, parser, and code generator all need to be tweaked a bit.

First, let's make sure that the tokenizer can recognize the token return and represent it with a token of type TK_RETURN. Since there are only a limited number of tokens (called keywords) that have special meaning in the syntax, such as return, while, and int, it is easier to use a different type for each token.

It seems that to check if the next token is a return, we only need to check if the rest of the input string of the tokenizer starts with return, but that would mean that a token like returnx would be tokenized as return and x by mistake. Therefore, here, in addition to making sure that the input starts with return, we also need to make sure that the next character is not one of the characters that make up the token.

Below is a function to determine if a given character is a token constituent character, i.e., alphanumeric or underscore.

```
int is_alnum(char c) {
 return ('a' <= c && c <= 'z') ||
          ('A' <= c && c <= 'Z') ||
          ('0' <= c && c <= '9') ||
          (c == '_');
}
```

Using this function, you can tokenize return as TK_RETURN by adding the following code to tokenize.

```
if (strncmp(p, "return", 6) == 0 && ! is_alnum(p[6])) {
 tokens[i].ty = TK_RETURN;
 tokens[i].str = p;
 i++;
 p += 6;
 Continue;
}
```

Next, let's modify the parser so that it can parse a token sequence containing TK_RETURN. To do so, we will first add the node type ND_RETURN, which represents the return statement. Next, we will modify the function that reads the statement so that it can parse the return statement. As usual, we can parse the grammar by mapping it directly to a function call. The new stmt function is shown below.

```
Node *stmt() {
 Node *node;

 if (consume(TK_RETURN)) {
     node = calloc(1, sizeof(Node));
     node->kind = ND_RETURN;
     node-> lhs = expr();
 } else {
     node = expr();
 }

 if (!consume(';'))
 error_at(tokens[pos].str, "';' is not a token");
 return node;
}
```

Since nodes of type ND_RETURN are only generated here, we decided to malloc and set the value on the fly instead of creating a new function here.

Finally, we will modify the code generator to output the appropriate assembly code for nodes of type ND_RETURN. A part of the new gen function is shown below.

```
void gen(Node *node) {
 if (node->kind == ND_RETURN) {
     gen(node-> lhs);
 printf(" pop rax\n");
 printf(" mov rsp, rbp\n");
 printf(" pop rbp\n");
 printf(" ret\n");
 return;
 }
 ...
```

The function call gen(node-> lhs) in the above code will output the code for the expression that is the return value of return. That code should leave a single value on the top of the stack, and in the assembly that follows gen(node->lhs), that value is popped from the stack, set to RAX, and returned from the function.

In the functions implemented up to the previous chapter, one ret instruction was always output at the end of the function. If we implement the return statement in the way described in this chapter, each return statement will output an extra `ret` instruction. It is possible to combine these instructions, but for the sake of simplicity, we have decided that it is acceptable to output multiple `ret` instructions. There is no point in worrying about these details at this point, so it is important to prioritize the simplicity of the implementation. Being able to write difficult code is a useful skill, but sometimes it's even more useful to not make the code too difficult in the first place.

### Column: Grammar Hierarchy

Regular expressions are often used to determine whether the input conforms to some rule, but more complex grammars cannot be expressed with regular expressions. For example, a regular expression that judges whether parentheses are balanced in a string cannot, in principle, be written.

Context-free grammars (grammars that can be expressed in BNF) are more powerful than regular expressions, and can, for example, represent only strings with balanced parentheses (written in BNF: `S → SS | "(" S ")" | ε`). However, like regular expressions, context free grammars have their limitations, and context free grammars cannot express the complex rules that appear in ordinary programming languages. For example, the rule "variables must be declared before use" is part of the C grammar, but such a rule cannot be expressed in context-free grammar.

If you write a C compiler, unless there is a bug in the compiler, you can say that "input accepted by the compiler is a correct C program, and input not accepted is an incorrect C program. In other words, if you have the ability of an ordinary computer, you can determine the question of "whether or not it matches the C grammar," and the compiler as a whole is a more powerful grammar-determining machine than a context-free grammar. In this way, a grammar that can always be judged as YES/NO to whether or not it matches the grammar is called decidable.

We can also consider grammars that are not Decidable. For example, it has been proven that it is generally impossible to determine yes or no to the question, "Given a computer program as input and executing it, will the program eventually `exit with an exit` function or will it continue executing indefinitely? It has been proven that it is generally impossible to determine yes or no without actually executing the program (assuming that it is executed on a virtual computer with infinite memory). In other words, we can answer YES when the program stops, but we cannot answer NO when it does not stop, because it will just keep executing indefinitely. This category of grammars in which the decision machine may not only return YES/NO, but the execution of the decision machine may never end, is called turing-recognizable.

In other words, there is a hierarchy of grammars: regular expressions < context-free grammars < decidable < turing-recognizable. This hierarchy of grammars has been widely studied as part of computer science. The famous unsolved problem P$\overset{?}{=}$NP is also a problem related to the hierarchy of grammars.

## C compiler in 1973

Up to this point, we have been building compilers incrementally. In a sense, this development process follows the history of C as it is.

If you look at C today, you will find that there are many parts that do not make sense or are unnecessarily complicated, but these cannot be understood without history. If you read the early C code and look at the early form of C and the subsequent development of the language and compiler, some of the things that are puzzling about C today will become clearer to you.

C was first developed in 1972 as a language for Unix, and the source code from 1972 or 1973, the very beginning of C's history, is still on tape, and files read from it are available on the Internet. Let's take a peek at some of the Ccompiler codefrom that time. The following is a function that receives a message in `printf` format and prints it as a compile error message.

```
error(s, p1, p2) {
 extern printf, line, fout, flush, putchar, nerror;
 int f;

 nerror++;
 flush();
 f = fout;
 fout = 1;
 printf("%d: ", line);
 printf(s, p1, p2);
 putchar('\fn');
 fout = f;
}
```

It looks somewhat strange, like a C-like language but not a C-like language. This was the kind of language C was back then. The first thing you notice when you read this code is that the return values and arguments of the functions have no type, just like in the early days of our compilers. Here, s is a pointer to a string, and p1 and p2 are supposed to be integers, but on the machines of the time, everything was the same size, so the variables are thus untyped.

The second line contains the declaration of the global variables and functions that error refers to. At that time, C compilers did not have header files or a C preprocessor, so the programmer had to tell the compiler about the existence of variables and functions in this way.

Just like our current compilers, functions are only checked to see if the name exists, not if the type or number of arguments match. After putting the expected number of arguments on the stack, you can just jump to the function body and the function call will succeed.

fout is a global variable that holds the file descriptor number of the output destination. At that time, fprintf did not yet exist, and in order to write strings to standard error output instead of standard output, it was necessary to switch the output destination via a global variable.

In error, printf is called twice, and in the second printf, two values are passed in addition to the format string. So what would you have done if you wanted to print an error message that only takes one value?

In fact, the ERROR function will work correctly even if you simply force it to read with fewer arguments. Recall that the argument checking of the function did not exist at this point; arguments such as s, p1, and p2 simply point to the first, second, and third words from the stack pointer, and the compiler does not care whether or not a value equivalent to p2 is actually passed. printf only accesses the extra arguments for the number of %d and %s in the first argument string, so if the message contains only one %d, p2 will not be accessed at all. Therefore, it does not matter if the number of arguments do not match.

As you can see, the early C compiler has many similarities to 9cc at this point in time.

Let's look at one more code example. The code below is a function that copies the passed string to a statically allocated area and returns a pointer that points to the beginning of the area. In other words, this is a function like strdup that uses a static region.

```
copy(s)
char s[]; {
 extern tsp;
 char tsp[], otsp[];

 otsp = tsp;
 while(*tsp++ = *s++);
 return(otsp);
}
```

At that time, the syntax for declarations of the form `int *p` had not been invented. Instead, pointer types are declared as `int p[]`. Between the function argument list and the function body, there is a kind of variable definition, which is used to declare `s` as a pointer type.

There are other things worth mentioning about this early C compiler.

- At this point, the structure did not exist.
- Operators such as `&&` and `||` are not yet available. In those days, `&` and`|` were context-sensitive, meaning that they became logical operators only in conditional expressions such as if.
- Operators such as `+=` were written as `=+`. There was a problem with this syntax that if you wrote `i=-1` without any spaces to assign -1 to `i`, it would be considered as `i =- 1` and `i` would be decremented, which was not the intended behavior.
- The only integer types were char and int, and there were no short or long types. There was no syntax for declaring types, such as "array of function pointers", and it was not possible to describe complex types.

In addition to the above, C in the early 70's lacked many other features. Nevertheless, this C compiler was written in C, as you can see from the source code above. C was already self-hosting at a time when there were not even structs.

Parsing variable definitions is easy if the syntax is always `extern`, `auto`, `int`, or `char` followed by the variable name. If the grammar is such that the variable name always comes after extern, auto, int, or char, the variable definition is easy to parse, and the pointer `[]` is also easy to parse if it just comes after the variable name. However, if this grammar is developed along the lines of what we see in this early compiler, I think we can understand why it has become unnecessarily complicated as it is today.

Now, what Dennis Ritchie, the co-developer of Unix and C, was doing around 1973 was truly incremental development. He was using C to write its compiler in parallel with the development of C itself. C as we know it today is not some finished product that has reached a particular point in the continuous addition of features to the language, it is simply a language that has reached a point where Dennis Ritchie felt that it had enough features to be a complete language.

Even our compiler did not pursue perfection from the beginning, because perfection in C is not particularly meaningful, and it would not make that much sense to specifically pursue it. Continuing to develop the language as a language with a reasonable set of features at any point in time, and eventually making it C, is the venerable development method that the primitive C compiler did. Let's proceed with development with confidence!

### Column: Rob Pike's Five Rules of Programming

9cc is influenced by the programming philosophy of Rob Pike, a former colleague of C author Dennis Ritchie, author of the Go language, and developer of UTF-8 for Unicode with Unix author Ken Thompson.

I quote from Rob Pike's 5 Rules of Programming.

1. There is no way to predict which parts of your program will use the most time. Bottlenecks occur in the most surprising places, so don't try to predict the location of a bottleneck and add performance hacks until you know where it is.
2. Measure. Don't try to optimize before you measure. And even if you do, don't try to optimize anything but the slowest parts of the code.
3. Elaborate algorithms are slow when n is small; n is usually small. Elaborate algorithms have a large constant part; don't elaborate unless you know that n is usually large. Don't elaborate unless you know that n is usually large. (Even if n is large, apply rule 2 first.
4. Elaborate algorithms are more buggy and difficult to implement than simple ones. Simple algorithms and data structures should be used.
5. Data is what matters. If you can choose the right data structure and represent the data well, the algorithm will almost always be self-evident. The data structure, not the algorithm, is the central entity of programming.

Step 12: Add the control syntax

*The chapters after this are still being written. The chapters up to this point have been carefully written, but to be honest, I don't think the chapters from here on are ready for publication. However, if you have been reading up to* *this point, you may be able to supplement what you need to read on your own, and some of you may want a guideline on how to proceed.*

In this section, we will add control structures such as if, if ... In this section, we will add control structures such as if, if ..., else, while, and for to the language. These control structures may look complex at first glance, but they are relatively easy to implement if you compile the assembly straight out of the box.

Since there is no corresponding C control structure in the assembly, C control structures are represented in the assembly by branch instructions and labels. In a sense, this is the same as rewriting the control structure using goto. Just as a human can rewrite a control structure into a goto statement by hand, a control structure can be implemented without difficulty by simply following a pattern to generate code.

There are other control syntaxes such as do ... There are other control syntaxes such as while, goto, continue, break, etc., but they do not need to be implemented yet at this point.

The new syntax with the addition of if, while, and for is shown below.

```
program = stmt*
stmt = expr ";"
         | "if" "("expr ")" stmt ("else" stmt)?
         | "while" "("expr ")" stmt
         | "for" "(" expr? ";" expr? ";" expr? ")" stmt
         | ...
...
```

When reading expr? ";", you can read one token ahead, and if the next token is ;, then the expr does not exist, otherwise you can read the expr.

if (A) B compiles into an assembly that looks like this

```
 Code that compiles A // The stack top should contain the result
 pop rax
 cmp rax, 0
 je . LendXXX
 Code compiled with B
. LendXXX:
```

That is, if (A) B

```
 if (A == 0)
 goto end;
 B;
end:
```

XXX should be a serial number so that all labels are unique.

if (A) B else C compiles into an assembly that looks like this

```
 Code that compiles A // The stack top should contain the result
 pop rax
 cmp rax, 0
 je . LelseXXX
 Code compiled with B
 jmp . LendXXX
. LelseXXX
 Code compiled with C
. LendXXX
```

In other words, if (A) B else C expands as follows

```
 if (A == 0)
 goto els;
 B;
 goto end;
els:
 C;
end:
```

When reading an if statement, we do a one token look ahead to see if there is an else, and if there is an else, we compile it as if ... If there is an else, it is compiled as an if without an else.

while (A) B compiles as follows

```
. LbeginXXX:
 Code that compiles A
 pop rax
 cmp rax, 0
 je . LendXXX
 Code compiled with B
 jmp . LbeginXXX
. LendXXX:
```

In other words, while (A) B will be expanded in the same way as the following code.

```
begin:
 if (A == 0)
 goto end;
 B;
 goto begin;
end:
```

`for (A; B; C) D` compiles as follows

```
 Code that compiles A
. LbeginXXX:
 Code compiled with B
 pop rax
 cmp rax, 0
 je . LendXXX
 Code compiled with D
 Code compiled with C
 jmp . LbeginXXX
. LendXXX:
```

`for (A; B; C) The` C code corresponding to `D` is shown below.

```
 A;
begin:
 if (B == 0)
 goto end;
 D;
 C;
 goto begin;
end:
```

`L is a` name that is specially recognized by the assembler, and automatically becomes file scope. L is a name that is specially recognized by the assembler and automatically becomes file-scoped. Therefore, if you make the compiler-generated labels for `if` and `for` start with `.L`, `you` don't have to worry about conflicts with labels contained in other files.

You can compile a small loop in cc and use that assembly as a reference to make it.

### Column: Compiler detection of runtime errors

When writing programs in C, it is common to write data beyond the end of an array, or to break unrelated data structures due to pointer bugs. These bugs can also be security holes, so the idea is to proactively detect bugs at runtime with the help of the compiler.

For example, if you pass the option `-fstack-protector` to GCC, the compiled function will output a pointer-sized random integer called a "canary" to the function frame in the prologue, and check that the value of the canary has not changed in the epilogue. In the epilogue, we will check that the value of the canary has not changed. In this way, if the contents of the stack have been overwritten unknowingly due to a buffer overflow of the array, the value of the canary will almost certainly have changed as well, which means that an error can be detected upon function return. When an error is detected, the program will usually terminate immediately.

LLVM has TSan (ThreadSanitizer), which can output code to detect at runtime if multiple threads are accessing a shared data structure without properly allocating a lock. Also, LLVM's UndefinedBehaviorSanitizer (UBSan) can output code to detect at runtime if an undefined behavior of C has been inadvertently stepped on. For example, since signed integer overflow is an undefined behavior in C, UBSan will report an error when a signed integer overflow occurs.

TSan, for example, slows down a program by several times, so it is unreasonable to add it as a compile option for programs that you use regularly. However, features such as stack canaries, which have relatively low runtime costs, may be turned on by default in some environments.

Such dynamic error detection with the help of compilers has been actively researched in recent years, and has greatly contributed to writing reasonably secure programs using languages such as C and C++, which are not memory-safe.

## Step 13: Block

This step supports "blocks" that can be written between `{ ...` `This step` supports "blocks" (blocks) that allow multiple statements to be written between `}`. A block is formally called a "compound statement", but since it is a long word, it is often referred to simply as a block.

Blocks have the effect of grouping multiple statements together into a single statement. The `if` and `while` implemented in the above step allowed only one statement to be executed when the conditional expression is satisfied, but by implementing blocks in this step, we can write multiple statements wrapped with `{}` there, just like in C.

The body of a function is also actually a block. Grammatically, the body of a function must be a block. The definition of a function `{ ... }` `in the function definition is` actually the same syntactically as `{ ... }` after an `if` or while. } in the function definition is actually syntactically the same as { ...

The syntax with the added blocks is shown below.

```
program = stmt*
stmt = expr ";"
         | "{" stmt* "}"
         | ...
...
```

To parse stmt* "}", call stmt repeatedly in a while statement until "}" appears, and then return the result as a vector. .

In order to implement a block, you need to add a node type ND_BLOCK to represent the block. In the structure Node representing the node, you need to add a vector with the expression contained in the block. In the code generator, if the type of the node is ND_BLOCK, make sure to generate the code for the statements contained in the node in order. Note that each statement leaves one value on the stack, so don't forget to pop it each time.

## Step 14: Respond to function calls

The goal of this step is to be able to recognize a function call with no arguments such as foo() and compile it to call foo.

The new syntax with the addition of function calls is shown below.

```
...
primary = num
         | ident ("(" ")")?
         | "("expr ")""
```

By reading one token ahead after reading ident, you can tell whether the ident is a variable name or a function name.

For testing, prepare a C file like int foo() { printf("OK\n"); }, compile it into an object file with cc -c, and link it with the output of your compiler. That way, you should be able to link the whole thing properly and make sure that the function you want to call is called properly.

Once you have that working, you can write function calls like `foo(3, 4)`. There is no need to check the number or type of arguments. You don't need to support more than six arguments.

For testing, you can prepare a function like `int foo(int x, int y) { printf("%d\n", x + y); }` in the same way as above, and link it to see how it works.

The ABI for function calls in x86-64 is easy (as long as you do it the way I did above), but there is one caveat. The RSP must be a multiple of 16 before the function call is made. `push` and `pop` change the RSP in units of 8 bytes, so the RSP is not necessarily a multiple of 16 when the call instruction is issued. If this promise is not kept, functions that assume that the RSP is a multiple of 16 will be plagued by a mysterious phenomenon that causes them to fail half the time. Make sure to adjust the RSP before calling the function, and adjust the RSP to be a multiple of 16.

### Step 15: Respond to the function definition

The next step is to make function definitions available. However, function definitions in C are a pain to parse, so we won't implement them all at once. Currently, our language has only the int type, so instead of using the syntax `int foo(int x, int y) { ... }` instead of the syntax `foo(x, y) { ... }` syntax with the type name omitted.

The caller needs to be able to access the argument by name, such as `x` or `y`. However, it is currently not possible to access the value passed in a register by name. So what to do is to compile the function as if local variables such as `x` and `y` existed, and in the prologue of the function, write the value of the register to an area on the stack for the local variable. After that, you should be able to handle arguments and local variables without any distinction.

Until now, the entire code was implicitly enclosed in `main() { ... }`, but `we'll do` away with that and write the whole code inside some function. That way, when you are parsing the top level, if you read the token first, it should always be the function name, followed by the argument list, followed by the function body, so it is easy to read.

After this step, you will be able to display the Fibonacci sequence while calculating it recursively, which should make things much more interesting.

## Binary level interface

The C language specification defines the source code level specifications. For example, the language specification determines how to define functions, which files to include, which functions to declare, and so on. On the other hand, the language specification does not specify what kind of machine language a source code written in compliance with the standard will be converted to, which is not surprising since the C language standard is not defined with a specific instruction set in mind.

Therefore, at first glance, it may seem that there is no need to properly determine machine language level specifications, but in reality, each platform has its own specifications to some extent. This specification is called ABI (Application Binary Interface).

In the function calls described so far in this book, the arguments were supposed to be placed in registers in a specific order. Also, the return value is promised to be set to RAX. These rules for calling functions are called "function calling convention". Function calling conventions are a part of ABI.

The ABI in C language includes the following as well as passing arguments and return values.

- Registers that are modified by the function call and registers that are not (RBPs, etc. are returned to their original values before return, but some registers do not need to be returned to their original values)
- Size of types such as `int` and `long`
- Rules for structure layout (rules for how the members of a structure are actually arranged in memory)
- Rules for bitfield layout (e.g., whether to arrange bitfields starting with the least significant bit or the most significant bit, etc.)

Since ABI is just a promise at the software level, it is possible to come up with something different from what is described in this book, but basically, CPU vendors and OS vendors define platform-standard ABI, since ABI-incompatible codes cannot be called and used by each other. In x86-64, there are two widely used ABIs: the System V ABI used in Unix and macOS, and the Microsoft ABI used in Windows. These two calling conventions are not separated by necessity, but simply because different people have developed them separately.

So far in this book, we have done things such as calling functions from our own compiler that were compiled by another compiler. This was possible because the ABI of our C compiler and the other compiler were the same.

[redacted]

# Representation of integers in computers

At this point, it is important to understand how integers, especially negative integers, are represented in computers. In this chapter, we will explain how to represent unsigned numbers and how to represent signed numbers using "two's complement" (two's complement).

In this book, binary bit patterns are represented with the 0b prefix, separated by underscores every four digits to make them easier to read, as in 0b0001_1010. 0b prefix is actually a compiler-specific extension that can be used as is by many C compilers (although it cannot normally be included). The 0b prefix is actually a compiler-specific extension that can be used as-is by many C compilers (although the inclusion of underscores is usually not allowed).

## unsigned integer

The representation of an unsigned integer is the same as that of a regular binary number, just as a decimal number represents digits 1, 10, 100, 1000, ..., (i.e., digits $^{100}$, $^{101}$, $^{102}$, $^{103}$, ..., ...) starting from the last digit. (i.e., $^{20}$ digits, $^{21}$digits, $^{22}$digits, $^{23}$digits, ...……).

For example, the value of an unsigned integer represented by the bit pattern 0b1110 can be found by looking at the position of the bit that is set to 1. In this case, the second, third, and fourth digits, i.e., digits 2, 4, and 8, are set to 1, so 0b1110 represents 2 + 4 + 8 = 14. Some example figures are shown below.

Adding 1 to an unsigned integer will cycle through the values, as shown in the following graph. This is an example of a 4-bit integer.

When the result of an operation overflows and the result is different from the result when there are infinite bits, it is called "overflowing. For example, in an 8-bit integer, 1 + 3 does not overflow, but 200 + 100 and 20 - 30 do, resulting in 44 and 246, respectively. Mathematically speaking, this is the same as the remainder divided by $^{28}$ = 256.

### Column: An interesting bug caused by overflow

Numerical overflow can sometimes lead to unexpected bugs. Here is a bug that was found in the first version of the game Civilization.

Civilization is a strategy simulation game where you fight between civilizations and choose a player like Genghis Khan or Queen Elizabeth to either conquer the world or win the space race.

The bug in the original Civilization was that the non-violent Gandhi suddenly went nuclear. The reason for this was the logic that when a civilization adopts democracy, its aggressiveness is reduced by 2. In the original Civilization, Gandhi's aggression was 1, the lowest of all players, but as the game progressed and the Indian civilization adopted democracy, his aggression was reduced by 2 to 255 in the overflow, and Gandhi suddenly became an extremely aggressive player in the game. Gandhi suddenly became an extremely aggressive player in the game. And by that time, the game had usually progressed to the level where each civilization had nuclear weapons in terms of science and technology, resulting in Gandhi's behavior of suddenly launching a nuclear war on one turn. This "nuclear Gandhi" was rather amusing and became a staple in the Civilization series from then on, but in the first game this was an unintentional bug.

## signed integer

In signed integers, the most significant bit is given special treatment in the "two's complement" representation. In the n-digit integer in the two's complement representation, all digits except the nth digit represent the same number as in the unsigned case, but the rule is that only the most significant n-digit represents $^{-2n-1}$ instead of $^{2n-1}$.

Specifically, if we consider a 4-digit binary number, each digit and the number it represents are shown in the following table.

|  | 4 | 3 | 2 | 1 |
|---|---|---|---|---|
| For unsigned | 8 | 4 | 2 | 1 |
| With sign | -8 | 4 | 2 | 1 |

As with unsigned, the signed value that a bit pattern represents can be found by looking at the position of the bit that is set to 1. For example, if we consider 0b1110 as a four-digit signed integer, the second, third, and fourth digits, i.e., digits 2, 4, and -8, are set to 1, so 0b1110 represents 2 + 4 + (-8) = -2. Some example figures are shown below.

In this rule, unless the most significant bit is turned on, the number represented by a signed integer is the same as the number that interprets it as an unsigned integer; in the case of a 4-bit integer, 0 to 7 will be the same bit pattern, signed or unsigned. On the other hand, if the fourth bit is on, then the bit pattern represents a number between -8 and -1 (0b1000 to 0b1111). When the most significant bit is on, the number is negative, so the most significant bit is sometimes called the "sign bit".

Adding 1 to a signed integer will cycle through the values, as shown in the following graph. This is an example of a 4-bit integer.

When you understand the above rules, you will be able to explain the various seemingly strange behaviors of signed integers that you commonly see in programming.

Readers have probably experienced that when you add 1 to a signed integer, it goes from a large number to an extremely small number at the overflow. If you consider the two's complement representation, you can understand what exactly is happening. For example, in an 8-bit signed integer, the largest number is 0b0111_1111, or 127. If you add 1 to it, you get 0b1000_0000, which is -128 in two's complement representation. This is the largest negative number in absolute value.

If you return, say, -3 from `main` in a unary test, the exit code for the entire program would have been 253. This is because `main` set RAX to -3, i.e. 0b1111_1111_1111_1111_1111_1111_1111_1111_1101. This is because main sets RAX to -3, i.e., 0b1111_1111_1111_1111_1111_1111_1111_1101, while the receiver thinks that only the lower 8 bits of RAX are meaningful and treats them as unsigned integers, resulting in 0b1111_1101, i.e., 253, as the return value. The result is as if 0b1111_1101, or 253, were the return value.

In this way, what kind of number a certain bit pattern represents depends on the assumptions of the reader. For example, characters in a paper book are just ink stains, but they have meaning only because some people read them as text. In order for a number to be passed on, the person setting the value and the person reading it need to have the same way of interpreting it.

Note that in two's complement representation, the number of negative numbers that can be represented is one more than the number of positive numbers that can be represented. For example, in an 8-bit integer, -128 is representable, but +128 is just outside the representable range. The reason for this imbalance in the range of positive and negative numbers is due to the way the system works: there are 2n patterns that can be represented by n bits, which means there are always an even number of patterns, but if you assign one bit pattern for 0, you will be left with an odd number of patterns, so either the positive number or the negative number will be larger. This is the reason why the number of bit patterns is so large.

## code extension

In computers, the operation of stretching the width of the bits of a number often comes up. For example, if you want to read an 8-bit number from memory and set it in a 64-bit register, you need to extend the 8-bit value to 64 bits.

If we are dealing with unsigned integers, extending the value is easy, we can simply fill in the upper bits with zeros. For example, the 4-bit value 0b1110 = 14 can be extended to 8 bits to get 0b0000_1110 = 14.

On the other hand, when dealing with signed integers, if you fill in the upper bits with zeros, the number will change. For example, if you expand the 4-bit value 0b1110 = -2 to 8 bits and set it to 0b0000_1110, it will become 14. This is not even a negative number because there is no sign bit to begin with.

When extending a signed integer, it is necessary to fill the new higher bits with all ones if the sign bit is one, or all zeros if the sign bit is zero. This operation is called "sign extension". For example, if we signally extend the 4-bit value 0b1110 = -2 to 8 bits, we get 0b1111_1110 = -2, which shows that we have successfully extended the bit width.

With unsigned integers, you can think of it as an infinite sequence of zeros to the left of the number, which you take out when you expand it.

Similarly, with signed integers, we can think of the left side of the number as having an infinite sequence of values equal to the sign bit, which we take out when we expand it.

Thus, if you are trying to fit a number into a wider bit width, you need to be aware of whether the value you are dealing with now is unsigned or signed.

### Column: representation of negative numbers without sign extension

Two's complement is a widely used way to represent signed integers in computers, but it's not the only way to map positive and negative integers into bit patterns. For example, consider a negative binary number, which represents $(-2)^0$, $(-2)^1$, $(-2)^2$, ..., starting from the last digit. Here is a comparison table of the number each digit represents, in the case of 4 bits.

| | 4 | 3 | 2 | 1 |
|---|---|---|---|---|

| | | | | |
|---|---|---|---|---|
| unsigned | 8 | 4 | 2 | 1 |
| 2’s complement | -8 | 4 | 2 | 1 |
| minus binary number | -8 | 4 | -2 | 1 |

A 4-bit negative binary number can represent a total of 16 integers， from -10 to 5, as shown below.

| | |
|---|---|
| 5 | 0b0101 |
| 4 | 0b0100 |
| 3 | 0b0111 |
| 2 | 0b0110 |
| 1 | 0b0001 |
| 0 | 0.0000 |
| -1 | 0b0011 |
| -2 | 0b0010 |
| -3 | 0b1101 |
| -4 | 0b1100 |

| | |
|---|---|
| -5 | 0b1111 |
| -6 | 0b1110 |
| -7 | 0b1001 |
| -8 | 0b1000 |
| -9 | 0b1011 |
| -10 | 0b1010 |

Minus binary numbers, as you can see, have the disadvantage that they are difficult to handle, such as digit additions, and there are no zeros near the middle of the expressible range, but on the other hand, they have the interesting feature that they do not require a sign bit. However, it has the interesting feature that it does not require a sign bit. Therefore, when extending a minus binary number with a certain digit to a larger number of digits, the upper bits can always be filled with zeros.

As you can see, there are various ways to represent integers on a computer, not limited to two's complement, and two's complement is the easiest to handle in hardware, and is used in almost all existing computers.

## Sign reversal

The details of two’s complement representation are not necessarily necessary for compiler writing, but it is useful to learn some tricks related to two's complement representation. Here is a simple way to invert the positive and negative values of a number.

In the two’s complement representation, the positive and negative values of a number are inverted when you do the ″flip all bits and add 1" operation. For example, the procedure to find the bit pattern from 3 to -3 in an 8-bit signed integer is as follows

1. The number is expressed as a binary number; for 3, 0b0000_0011.
2. Invert all bits. In this case, the value is 0b1111_1100.
3. Add 1. In this case, it becomes 0b1111_1101. This is the -3 bit pattern.

Remembering the above method, you can easily find the bit pattern of a negative number.

Also, by making a bit pattern with a sign bit standing for a positive number by performing the same operation, we can sometimes easily find the number that the bit pattern represents. For example, it is tedious to find out what 0b1111_1101 represents by simply adding it up, but if you flip the bit and add 1, you get 0b0000_0011, which is easy to see that it represents the reverse sign of 3, or -3.

The reason why the above trick works is rather simple. Up to this point, I have not properly defined the operation of the two's complement representation mathematically, so the explanation is rather vague, but the idea is as follows.

Inverting all bits is the same as subtracting from a bit pattern with -1, i.e. all bits are 1. For example, the bit pattern 0b0011_0011 can be inverted as follows

```
  1111 1111
- 0011 0011
= 1100 1100
```

In other words, inverting the bit pattern representing the number n is the same as calculating -1 - n. If you add 1 to it, you are calculating (-1 - n) + 1 = -n. If you add 1 to it, you are calculating (-1 - n) + 1 = -n, which means that you have found -n for n.

### Column: The radix of the number of literals

In the C standard, numbers can be written in octal, decimal, or hexadecimal. Writing a number normally as 123 is decimal, writing it with 0x at the beginning as 0x8040 is hexadecimal, and writing it with leading zeros as 0737 is octal.

Many readers may think that they have never used the ability to write numbers in octal in C. However, in this syntax, a mere 0 is also octal notation, so all C programmers actually write octal numbers very frequently. This is a bit of trivia, but if you think about it, there is a deep and not-so-deep reason.

In the first place, 0 is a rather special notation for numbers. Normally, a number such as 1 is not written as 01 or 001 even though the tenth or hundredth digit is zero, but if we apply the same rule to 0, it will become an empty string. In this case, we have a special rule to write 0.

[redacted]

# Pointers and String Literals

In the previous chapters, we have created a language that can do some meaningful calculations, but our language is not even capable of displaying `Hello world` yet. It is time to add strings so that we can output meaningful messages from our programs.

String literals in C are closely related to the `char` type, global variables, and arrays. As an example, consider the following function.

```
int main(int argc, char **argv) {
 printf("Hello, world!\n");
}
```

The above code is to be compiled in the same way as the code below. However, ″`msg`″ should be a unique identifier that is not covered by any other identifier.

```
char msg[15] = "Hello, world!\n";

int main(int argc, char **argv) {
 printf(msg);
}
```

Our compiler is still lacking some features to support string literals. In order to support string literals and be able to print messages using `printf`, etc., let's implement the following features in order in this chapter.

1. Unary `& and` Unary*.
2. pointer
3. Array
4. Global variables

5. char type
6. string literal

We will also add the functions needed to test the above features and other functions in this chapter.

### Step 16: Unary & and Unary *.

In this step, as the first step to implement pointers, we implement unary & which returns an address and unary * which refers to an address.

These operators originally return or take the value of a pointer type, but since our compiler does not yet have any type other than integer, we will substitute the pointer type with an integer type. In other words, &x returns the address of variable x as a simple integer. In other words, &x returns the address of variable x as a simple integer, and *x reads the value of x from its address.

If we implement such an operator, the following code will work.

```
x = 3;
y = &x;
return *y; // return 3
```

You can also take advantage of the fact that local variables are allocated contiguously in memory to force indirect access to variables on the stack via pointers. The following code assumes that the variable x is 8 bytes above the variable y on the stack.

```
x = 3;
y = 5;
z = &y + 8;
return *z; // return 3
```

In such an implementation that does not distinguish between pointer and integer types, for example, the expression *4 would be an expression to read a value from address 4, but that's OK for now.

The implementation is relatively simple. The grammar with the addition of unary& and unary* is shown below. Follow this grammar to modify the parser to read unary& and unary* as nodes of type ND_ADDR and ND_DEREF, respectively.

```
unary = "+"? primary
```

```
         | "-"? primary
         | "*" unary
         | "&" unary
```

There are very few changes to make to the code generator. The changes are listed below.

```
 case ND_ADDR:
     gen_lval(node->lhs);
 return;
 case ND_DEREF:
     gen(node->lhs);
     printf(" pop rax\n");
     printf(" mov rax, [rax]\n");
     printf(" push rax\n");
 return;
```

## Step 17: Abolish the implicit variable definition and introduce the keyword int

Until now, the return value of any variable or function was implicitly assumed to be int. Therefore, we did not bother to define variables with their type names, such as `int x;`, and we assumed that every new identifier was a new variable name. From now on, we cannot make such an assumption. So, we will modify that first. Please implement the following features.

- Don't consider new identifiers as variable names, and make it an error if a variable appears that is not defined.
- Initialization expressions such as `int x = 3;` do not need to be supported. Initialization expressions such as int x = 3; need not be supported, and similarly `int x, y;` are not needed. Implement only the simplest things possible.
- The function used to be written as `foo(x, y)`, but we will modify it to be `int foo(int x, int y)`. Currently, there should only be function definitions at the top level, so the parser will first read the `int`, then the function name, which should always be the function name, and then the `int <argument name>` column. You don't need to deal with any more difficult syntax, and you don't need to do anything just in case you want to extend it in the future. Simply write enough code to read "int < function name>(<argument list consisting of repeated int < variable name>)".

Step 18: Introduce the pointer type

## Define a type to represent a pointer

In this step, we will allow type names that are an int followed by zero or more *'s, instead of only int. In other words, we will be able to parse definitions such as int *x and int ***x.

A type such as "pointer to int" must of course be handled by the compiler. For example, if the variable x is a pointer to int, then the compiler must know that the expression *x is of type int. This cannot be represented by a fixed size type, because the type can be as complex as "pointer to pointer to int".

So what do we do? We use pointers. Up until now, the only information that has been tied to a variable through a map is the offset from the base pointer (RBP) on the stack. We need to make a change to this so that we can have the type of the variable. The type of a variable, roughly speaking, should be a structure that looks like the following

```
struct Type {
 enum { INT, PTR } ty;
 struct Type *ptr_to;
};
```

Here, a ty can have one of two values: either an int type or a "pointer to ~" type. ptr_to is a member that only makes sense if the ty is a "pointer to ~" type, in which case it should contain a pointer to the Type object pointed to by "~". For example, if it is a "pointer to int", then the data structure representing that type would internally look like this

Data structure representing a pointer to an int

A "pointer to a pointer to an int" would look like this

Data structure representing a pointer to a pointer to an int

In this way, you can represent any number of difficult types inside the compiler.

## Assign to the value pointed to by the pointer

How do I compile an expression where the left side of the assignment expression is not a simple variable name, such as *p=3? The basic concept of such an expression is the same as when the left-hand side is a simple variable. In this case, we just need to compile *p as the left-hand side value so that the address of p is generated.

When compiling a syntax tree representing *p=3, we recursively descend the tree to generate code, and the first thing that is called is a code generator to compile with *p as the left-hand side value.

The code generator will branch depending on the type of the given syntax tree. For a simple variable, it will output the code that outputs the address of that variable, as mentioned above, but here, since the dereference operator is given, it should behave differently. If a dereference operator is given, compile the syntax tree in it as a "right-hand side value". Then it should compile into code that computes some address (otherwise you can't dereference the result). Then you can just leave the address on the stack.

Once you have completed this step, you should be able to compile a statement like the following

```
int x;
int *y;
y = &x;
*y = 3;
return x; // → 3
```

## Step 19: Implement pointer addition and subtraction

In this step, we will be able to write expressions such as p+1 and p-5 for a pointer type value p. This looks like just integer addition, but it is actually quite a different operation. This may look like a simple integer addition, but it is actually quite a different operation. p+1 does not mean to add 1 to the address that p has, but to make the pointer point to the next element of p, so we have to add the width of the data type that the pointer points to to p. So you have to add the width of the data type that the pointer points to to p. For example, if p points to int, then in our ABI, p+1 will add 4 bytes to the address. On the other hand, if p is a pointer to a pointer to an int, then p+1 will add 8.

Therefore, when adding and subtracting pointers, we need a way to know the size of the type, but currently it is 4 for int and 8 for pointer, so please write your code in such a deterministic way.

At this stage, there is no way to allocate memory continuously yet (our compiler doesn't have arrays yet), so it is a bit hard to write tests. Here you can simply take the help of an external compiler and malloc it on your side, and try to write tests using its helper functions in the output of your own compiler. For example, you could test something like this

```
int *p;
alloc4(&p, 1, 2, 4, 8);
int *q;
q = p + 2;
*q; // → 4
q = p + 3;
return *q; // → 8
```

### Column: Size of int and long

A data model such as x86-64 System V ABI, where int is 32 bits and long and pointer are 64 bits, is called LP64. This means that long and pointer are 64 bits. For the ABI on the same x86-64, Windows uses LLP64, a data model where int and long are 32 bits and long long and pointer are 64 bits.

LP64 and LLP64 are not ABI compatible because the size of long is different. For example, if you create a structure that contains long members, write the entire structure directly to a file, and when you read the file, cast the data on the file directly to the structure, you cannot pass the file to each other for reading on Unix and Windows.

The C specification states that an int is "the natural size of an integer on the machine" (A "plain" int object has the natural size suggested by the architecture of the execution environmen). However, what is natural is a subjective matter, and even on a 64-bit machine, 32-bit operations can usually be handled naturally, so it is not necessarily wrong to use 32-bit int on a 64-bit machine. It is not wrong to use 32-bit int on 64-bit machines. In addition, from a practical point of view, the following problems arise when using 64-bit int.

- Since there are very few cases where an int needs to be as large as 64 bits, making an int 64 bits is simply a waste of memory.
- If we assume that short is 16 bits and int and long are 64 bits, there will be no type to represent 32-bit integers.

For the reasons mentioned above, int is 32-bit on most existing 64-bit machines. However, there are some ILP64 machines with 64 bit int. For example, the old Cray supercomputer was an ILP64.

## Step 20: sizeof operator

Although `sizeof` looks like a function, grammatically it is a unary operator; in C most operators are symbols, but grammatically there is no particular reason why an operator should be a symbol, and `sizeof is actually an` exception to that rule.

Let's review a little bit how the `sizeof` operator works. `sizeof is` an operator that returns the number of bytes in memory of the type of the argument expression. For example, in our ABI, `sizeof(x)` will return 4 if `x` is an `int`, and 8 if `x` is a pointer. `sizeof`'s argument can be any expression, for example, `sizeof(x+3)` will return 4 if the expression `x+3 is an` int as a whole, and 8 if it is a pointer. For example, sizeof(x+3) will return 4 if the expression x+3 is an int or 8 if it is a pointer.

Our compiler does not have arrays yet, but `sizeof(x) will` return the size of the whole `x` as bytes if `x` is an array. For example, if `x` is defined as `int x[10]`, then `sizeof(x)` will return 40. if `x` is defined as `int x[5][10]`, `then sizeof(x)` will be 200, `sizeof(x[0])` will be 40, and `sizeof(x[0][0])` will be 4.

The argument of the `sizeof` operator is only written to know the type, not the actual expression to be executed. For example, if you write the expression `sizeof(x[3])`, no access to `x[3] will actually` occur; the overall type of the expression `x[3] will be` known at compile time, and the expression `sizeof(x[3]) will be` replaced by the size of that type at compile time. Therefore, the specific expression given to `sizeof`, such as `x[3]`, will no longer exist at runtime.

The behavior of `sizeof` is shown below.

```
int x;
int *y;

sizeof(x); // 4
sizeof(y); // 8

sizeof(x + 3); // 4
sizeof(y + 3); // 8
sizeof(*y);    // 4

// Any expression can be passed to sizeof
sizeof(1); // 4

// The result of sizeof is the same as sizeof(int) because it is now of
type int
sizeof(sizeof(1)); // 4
```

Now, let's try to implement the `sizeof` operator, which requires modifications to both the tokenizer and parser.

First, make a change to the tokenizer so that it recognizes the keyword `sizeof` as a token of type `TK_SIZEOF`.

Next, we will make a change to the parser to replace `sizeof` with a constant of type `int`. The grammar with the addition of the `sizeof` operator is shown below. In the following grammar, `sizeof` is defined as a unary operator, which has the same precedence as unary plus and unary minus. This is the same as the C grammar.

```
unary = "sizeof" unary
      | ("+" | "-")? primary
```

In this grammar, it is grammatically permissible to write something like `sizeof x` instead of just `sizeof(x)`, and it is the same in real C.

In the parser, when the `sizeof` operator appears, parse the expression that is its argument as usual and replace it with the number 4 if the type associated with the resulting parse tree is `int`, or 8 if it is a pointer. There is no need to make any changes to the code generation tree since the parser will replace it with a constant.

Step 21: Implement the array

# Define the array type

In this step, we will implement arrays. Up to this point, we have only dealt with data that can fit into registers, but this is the first time we will be dealing with data larger than that.

However, the C syntax is restrained when it comes to arrays. You cannot pass an array as an argument to a function, nor can you return an array as the return value of a function. If you write code with this intention, a pointer to the array will be automatically created and passed to you, rather than the array itself being passed by value. Directly assigning an array to an array and copying it is also not supported (you have to use memcpy).

The ability to allocate an area of memory larger than one word on the stack is sufficient.

Make sure that the following variable definitions are readable.

```
int a[10];
```

The type of a above is an array, which has a length of 10 and an element type of int. As with pointer types, array types can be as complex as you want, so as in step 7, the type of the array elements should be pointed to by ptr_to. The structure representing the type should look like this

```
struct Type {
 enum { INT, PTR, ARRAY } ty;
 struct Type *ptr_to;
 size_t array_size;
};
```

where array_size is a field that is only meaningful for array types, and is a variable that has the number of elements in the array.

If you can get this far, it should be easy to allocate an area on the stack for the array. To get the size of the array in bytes, just multiply the size of the array elements in bytes by the number of elements in the array. Up to now, you would have allocated the stack area as one word for all variables, but change that and allocate the size of the array as required for the array.

## Implement implicit type conversion from arrays to pointers

Since arrays and pointers are often used in combination, the syntax of C allows pointers and arrays to work without much distinction, but this backfires and makes it difficult for programmers to understand the relationship between arrays and pointers. In this section, I will explain the relationship between arrays and pointers.

First, in C, arrays and pointers are completely separate types.

A pointer is (in x86-64) an 8-byte value type, and just as the + and - operators are defined for int, they are defined (somewhat differently) for pointers. In addition, pointers have the unary* operator defined, which can be used to refer to the destination of a pointer. Apart from unary*, there is nothing special about pointers. In other words, a pointer is just an ordinary type like an int.

Arrays, on the other hand, are types that can be any number of bytes. Unlike pointers, there are few operators defined for arrays. The only operators that are defined are the sizeof operator, which returns the size of the array, and the & operator, which returns a pointer to the first element of the array. Other than that, there are no other operators that can be applied to arrays.

So why is it possible to compile an expression like a[3]? In C, a[3] is defined as being equivalent to *(a+3). I thought the + operator was not defined for arrays?

This is where the syntax of arrays being implicitly converted to pointers comes into play, as arrays are implicitly converted to pointers that point to the first element of the array, except when used as sizeof or unary & operands. Thus, *(a+3) is an expression that dereferences the pointer to the first element of array a plus 3, which is equivalent to accessing the third element of the array.

In C, there is no [] operator for array access; C's [] is just a convenient notation for accessing array elements via pointers.

Similarly, passing an array as a function argument can be written as a pointer to the first element of the array, or as if the array is directly assigned to the pointer, for the same reason as above.

So the compiler has to do a type conversion of the array to a pointer in most of the operator implementations. This should not be too difficult to implement, unless you are implementing sizeof and unary&, in which case you can just parse the operand of the operator to be a pointer to T if the type is an array of T. In the code generator, you should be able to generate code that says that the value of an array type pushes the address of the value onto the stack.

If you have completed this far, you should be able to run the following code.

```
int a[2];
*a = 1;
*(a + 1) = 2;
int *p;
p = a;
return *p + *(p + 1) // → 3
```

### Column: Language Lawyer

A person who has a good understanding of a formal language specification is sometimes called a ″language lawyer,″ using the language specification as a law. The Jargon File, a dictionary of programmer slang, describes a language lawyer as follows:[6]

- Language lawyer [noun]: an experienced or senior software engineer who is familiar with (almost) all the useful and strange features and limitations of one or more programming languages. You can tell if someone is a language lawyer by whether he or she can answer a question by showing five sentences scattered throughout a 200+ page manual and saying, ″You should have looked here.

The word ″language lawyer″ can also be used as a verb to describe language lawyering.

It seems that skilled language lawyers are often looked up to by other programmers. When I was working on the C++ compiler team at Google, there was a guy on the team who was the ultimate language lawyer, and we would often conclude that if we didn't understand something in C++, we should ask him (there are many things in the C++ specification that even the people who make C++ compilers don't understand). ). In fact, he was the person who implemented the main part of Clang, a major C++ compiler, and was also the first author of the C++ specification, and was one of the most knowledgeable people in the world about C++. I remember thinking that the enormity of the C++ language specification and the complexity of the details are quite something.

In this book, we intentionally do not go into the details of the C language specification in depth until the compiler is complete. There is a reason for that. When implementing a programming language for which specifications exist, it is necessary to become a language lawyer to some extent, but it is undesirable as a development technique to worry too much about the details from the beginning. Just as when you draw a picture, you should first complete a rough sketch of the whole picture instead of drawing only one part in detail, when you are implementing a programming language, you need to maintain a balance and not be too much of a language lawyer at first.

## Step 22: Implement array indexing

In C, `x[y]` is defined as being equivalent to `*(x+y)`. Therefore, the implementation of subscripts is relatively simple. Simply rewrite `x[y]` as `*(x+y)` in the parser. For example, `a[3]` becomes `*(a+3)`.

In this grammar, `3[a]` expands to `*(3+a)`, so if `a[3]` works, then `3[a]` should work too, but to my surprise, expressions like `3[a]` are actually legal in C. Try it out.

## Step 23: Implement the global variables

In C, a literal string is an array of char. In C, a literal string is an array of char, which is fine because we already implement arrays, but the difference is that a literal string is not a value that exists on the stack. A string literal is not on the stack, but in a fixed location in memory. Therefore, to implement a string literal, we will first add a global variable.

Until now, only function definitions were allowed at the top level. We will change that syntax to allow global variables to be written at the top level.

Variable definitions are somewhat tricky to parse because they look similar to function definitions. For example, compare the following four definitions.

```
int *foo;
int foo[10];
int *foo() {}
int foo() {}
```

The top two foo's are variable definitions and the bottom two are function definitions, but the two are indistinguishable until you get to the identifier that becomes the function or variable name, and then read the next token. Therefore, we need to call the "read the first half of the type name" function first, and then read the identifier, which should come after the first half of the type name, and then try to read one of the tokens first. If the token you read is "(", you are reading the function definition, otherwise you are reading the variable definition.

The names of the parsed global variables should be put in a map so that they can be looked up by name. Only if the variable name cannot be resolved as a local variable will it attempt to be resolved as a global variable. This allows for the natural implementation of a local variable hiding a global variable of the same name.

In the parser, references to local variables and references to global variables are converted to different nodes in the abstract syntax tree. Since the names can be resolved at the parsing stage, the types are also separated at that stage.

Until now, all variables were supposed to be on the stack, so reading and writing variables was done relative to the RBP (base pointer). Since a global variable is not a value on the stack, but a value at a fixed location in memory, we can compile the code to access that address directly. Please refer to the actualgccoutput.

When you implement it, you will be surprised to find that local variables and global variables are quite different. The reason why you can write them without apparent distinction is because the C language abstracts them well. Local variables and global variables are actually implemented quite differently internally.

## Step 24: Implement the character type

Arrays were types that could be larger than one word, but characters are types that are smaller than one word. Before you got to this step, you probably had to write a function that takes an object representing a type and returns the number of bytes of the size of that type. First, you need to add the type character, and then make changes to that function so that it returns 1 for the character type.

There is no need to implement literal characters (characters enclosed in single quotes) in this step. Resist the urge to implement them all at once and keep the changes as small as possible.

Thus, at this step, a character is really just a small integer type. `movsx ecx, BYTE PTR [rax]` will read one byte from the address pointed to by RAX and put it into ECX. If you do not need sign extension, use the `movzx` instruction as `movzx ecx, BYTE PTR [rax]`. When writing out, use an 8-bit register as the source register, such as `mov [rax], cl`.

Please refer to the actual compiler output.

If we can implement this step, we should be able to get the following code working.

```
char x[3];
x[0] = -1;
x[1] = 2;
int y;
y = 4;
return x[0] + y; // → 3
```

### Column: Difference between 8-bit registers and 32-bit registers

Why do I need to use movsx or movzx when reading a 1-byte value? When reading a 4-byte value, I could just load it into a lower 32-bit alias register like EAX with a regular mov, so when reading a char, I should just load it into AL with a regular mov. However, that doesn't work properly. The answer to this mystery lies in the x86-64 specification.

In x86-64, the upper 32 bits are reset to 0 when loaded into the lower 32 bits of the alias register. However, when reading into the lower 8-bit alias register, the upper 56 bits remain at their previous value. This is an inconsistent specification, but since x86-64 is an instruction set with a long history, such inconsistencies exist in many places.

Since x86-64 evolved from a 16-bit processor called 8086 to 32-bit and 64-bit, there was first AL, then EAX, and then RAX. In other words, the specification originally existed that the upper 24 bits of EAX would not be reset when loaded into AL (they would remain unchanged), and when it was extended to 64 bits, the specification was formulated that the upper 32 bits of RAX would be reset when loaded into EAX. There is a good reason why this kind of inconsistent specification was chosen.

Modern processors look at instruction dependencies and execute unrelated instructions (instructions that do not use the result of the previous instruction) in parallel. Assuming that the instruction set does not reset the upper 32 bits, if the programming style is such that the upper 32 bits are also present as garbage, but are ignored until the end and only the lower 32 bits are used, the instruction that generated the ignored upper 32 bits and the instruction that uses the same register that follows will be executed in parallel. If the programming style is to ignore the upper 32 bits until the end and continue to use only the lower 32 bits, a false dependency will be created between the instruction that generated the ignored upper 32 bits and the subsequent instruction that uses the same register. If the upper 32 bits are sign-extended and reset, the previous value will be completely overwritten, thus breaking the dependency. This is why when we converted x86 to 64 bits, we decided on a specification that was easy to speed up, knowing that it would compromise consistency.

## Step 25: Implementing String Literals

In this step, we will parse the double-quoted string so that it can be compiled. Now that we have the necessary parts: an array, a global variable, and a character type, it should be relatively easy to implement.

First, get your hands on the tokenizer, and when you find a double-quote, read to the next double-quote to create a string token. There is no need to implement backslash escaping or anything like that in this step. It is important to go step by step, so even if it seems easy to implement, try not to.

The code of an assembly that represents string literal data cannot be output in the middle of generating code for a machine that will run on the CPU. In the output assembly, the global data and the code must be written without mixing them. In other words, when we output the code, we want to output all the string literals that appeared in the code first, but it is tedious to go down the syntax tree to do so. To do this, it would be easy to prepare a vector that contains all the string literals seen so far, and simply add to it whenever the parser sees a string.

Please refer to the actual compiler output.

At this point, you should be able to use `printf to` output strings. This is a good opportunity to use your own programming language to write more elaborate programs, rather than the obvious ones such as test codes. For example, a solver for the 8 queens problem could be written in your own language. It took mankind decades from the invention of the digital computer to the development of a programming language that could be easily coded at this level. The fact that it can be implemented in a few weeks is a wonderful advancement for mankind and for you.

(When you call a function that takes variable-length arguments, you are supposed to put the number of floating-point arguments in AL. Our compiler does not have floating point numbers yet. Therefore, we should always set AL to 0 before calling a function).

## Step 26: Read the input from the file

Up to this point, we have been passing the C code directly as the argument string, but since the input is getting longer and longer, it is time to modify it to take the file name as the command line argument like a normal C compiler. A function that opens a given file, reads its contents, and returns a string terminated with `'\0'` can be written concisely as follows

```
#include <errno.h>.
#include <stdio.h>.
#include <string.h>.

// return the contents of the specified file
char *read_file(char *path) {
 // Open the file
 FILE *fp = fopen(path, "r");
 if (!fp)
     error("cannot open %s: %s", path, strerror(errno));

 // find out the length of the file
 if (fseek(fp, 0, SEEK_END) == -1)
     error("%s: fseek: %s", path, strerror(errno));
 size_t size = ftell(fp);
 if (fseek(fp, 0, SEEK_SET) == -1)
     error("%s: fseek: %s", path, strerror(errno));

 // read the contents of the file
 char *buf = calloc(1, size + 2);
 fread(buf, size, 1, fp);

 // Make sure that the file always ends with "\n\0
 if (size == 0 || buf[size - 1] ! = '\n')
     buf[size++] = '\n';
 buf[size] = '\0';
 fclose(fp);
 return buf;
}
```

For the convenience of the compiler implementation, we decided to automatically add `\n` if the last byte of the file is not `\n`, since it is easier to handle data where every line ends with a newline character than data where every line ends with a newline character or EOF.

Strictly speaking, this function does not work well when a special file that is not randomly accessible is given. For example, if you specify the device file `/dev/stdin, which represents` the standard input, or a named pipe as the file name, you should see an error message such as `/dev/stdin: fseek: Illegal seek`. However, for practical use, this function will not be a problem. You can change your code to use this function to read the contents of the file and treat it as input.

Since input files usually contain multiple lines, let's also enhance the function to display error messages. If we decide to print the name of the input file, the line number of the line with the error, and the contents of that line when an error occurs, the error message will look like this

```
foo.c:10: x = y + + 5;
                   ^ Not a formula.
```

The function to display such an error message looks like this

```
// Input file name
char *filename;

// Function to report the location where the error occurred
// Display error messages in a format similar to the one below
//
// foo.c:10: x = y + + 5;
//                   an expression
void error_at(char *loc, char *msg) {
 // Get the start and end points of the line containing the loc
 char *line = loc;
 while (user_input < line && line[-1] != '\n')
     line--;

 char *end = loc;
 while (*end ! = '\n')
     end++;

 // Find out how many lines the found line is in the whole
 int line_num = 1;
 for (char *p = user_input; p < line; p++)
 if (*p == '\n')
       line_num++;

 // Display the lines found, along with their file names and line numbers
 int indent = fprintf(stderr, "%s:%d: ", filename, line_num);
 fprintf(stderr, "%.*s\n", (int)(end - line), line);

 // Display error messages by pointing to the error location with "^".
 int pos = loc - line + indent;
 fprintf(stderr, "%*s", pos, ""); // print pos spaces
 fprintf(stderr, "^ %s\n", msg);
 exit(1);
}
```

This error message output routine is pretty simple to make, but I guess you could say that it outputs errors in a pretty authentic looking format.

### Column: Error Recovery

If the input code is grammatically incorrect, many compilers will try to skip over the error and continue parsing. The purpose of this is to find as many errors as possible, not just one. The ability of the parser to recover from errors and continue parsing is called ″error recovery″.

In the 1960's and 1970's, programmers used to time-share large computers in computing centers, and they had to wait, sometimes overnight, from the time they brought in the code they wanted to compile until they got the results. In such an environment, it was necessary to point out the problem. In such an environment, one of the most important tasks for a compiler was to point out as many errors as possible. In older compiler textbooks, error recovery has become one of the main topics in parsing.

Nowadays, development with compilers is much more interactive, so error recovery is not such an important topic. In the compilers we develop, we only print the first error message. This is sufficient in many cases today.

## Step 27: Line Comments and Block Comments

Our compilers are getting better and better, and we can now write real code. In this case, we want comments. In this chapter, we will implement comments.

There are two types of comments in C. One comment is called a line comment, where the comment is from `//` to the end of the line. The other is called a block comment, where `/*` is the start symbol and `*/` is the end symbol. All characters in a block comment are skipped, except for the sequence of two characters, `*/`.

Grammatically, a comment is treated the same as a single space character. Therefore, it is natural to skip comments in the same way as whitespace characters in the tokenizer. The code to skip comments is shown below.

```
void tokenize() {
 char *p = user_input;

 while (*p) {
 // Skip whitespace characters
 if (isspace(*p)) {
       p++;
 Continue;
     }

 // Skip line comments
 if (strncmp(p, "//", 2) == 0) {
       p += 2;
 while (*p ! = '\n')
         p++;
 Continue;
     }

 // Skip block comments
 if (strncmp(p, "/*", 2) == 0) {
 char *q = strstr(p + 2, "*/");
 if (!q)
         error_at(p, "Comment not closed");
       p = q + 2;
 Continue;
     }
     ...
```

To find the end of the block comment, we used the `strstr` function in the C standard library. `strstr` looks for a string in a string and returns a pointer to the beginning of the passed string if it is found, or NULL if it is not found.

### Column: Line Comments

In the original C, only block comments existed, and line comments were officially added to the specification in 1999, almost 30 years after C was developed. The original intention was that this should have been a change that would not break compatibility, but in practice, in some subtle cases, the code that originally worked can mean something else.

Specifically, the following code will be loaded as `a/b` if only block comments are supported, and as `a if` line comments are supported.

```
a//*
// */ b
```

### Column: Block Comments and Nesting

Block comments are not allowed to be nested. `/*` has no special meaning in comments, so you can comment out existing block comments and use

```
/* /* ... */ */ ...
```

would cause the comment to end at the first `*/`, and the second `*/` would cause a syntax error.

If you want to comment out a group of lines that may contain block comments, you can use the C preprocessor to

```
#if 0
...
#endif
```

You can also wrap it with `#if 0`, as in

## Step 28: Rewrite the tests in C

In this step, we will rewrite the tests to speed up `make test`. By the time you get to this step, you probably have more than 100 tests written in your shell script. In shell script tests, many processes are launched per test. In other words, for each test, you are running your own compiler, assembler, linker, and the test itself.

Launching a process is not that fast, even for a small program. Therefore, if you do it hundreds of times, it will take a non-negligible amount of time in total. Your test script will probably take a few seconds to run.

The reason I was writing tests in shell scripts in the first place was because I could not do proper testing otherwise. At the stage of calculator-level languages, there were no `if`s, `==`s, etc., so you couldn't verify within that language whether the calculation results were correct or not. But now we can verify it. It is now possible to compare whether the result is correct or not, and if it is wrong, print an error message (in a string) and exit.

So, in this step, please rewrite the tests that were written in shell scripts into C files.

# Program execution image and initialization expression

Now, with the steps we have taken so far, our compiler now supports all the major elements of programming: functions, global variables, and local variables. We also hope that the readers have learned about split compilation and linking, which will help them understand how to compile a program in small pieces and finally combine them into a single file.

This chapter explains how the OS executes an executable file. By reading this chapter, you will be able to understand what kind of data is contained in the executable and what happens before the main function is called.

This chapter also explains variable initialization expressions, i.e. how code like the one below is compiled, and adds support for initialization expressions to our compiler.

```
int x = 3;
int y[3] = {1, 2, 3};
char *msg1 = "foo";
char msg2[] = "bar";
```

In order to support initialization expressions, it may come as a surprise, but the knowledge of how a program works to get to main is essential.

In this chapter, we will discuss a simple executable that contains all the code and data in a single executable file. Such an executable file is called a ″statically linked″ executable file. In contrast to static linking, dynamic linking, in which a program fragment is contained in multiple files that are combined in memory at runtime, is also widely used, but will be explained in a separate chapter. First, let's have a good understanding of the basic model, static linking.

## Structure of the executable file

An executable file consists of a file header and one or more areas called ″segments", where executable code and data are stored separately. A single executable file usually has at least two segments, one containing the executable code and one containing the data, called the ″text segment" and the other containing the data, called the "data segment". The actual executable file contains other segments as well, but they are not necessary to understand how it works, so they are omitted here.

Regarding terminology, please note that "text" is the same word as text in text files, but it has a different meaning. Traditionally, in the lower layers, data representing machine words is called "text". Also, machine language is just a sequence of bytes, and text is a kind of data, but when we talk about "text and data", "data" mostly refers to "data other than text". In this chapter, too, when we say data, we mean data other than text.

The object files that serve as input to the linker contain separate text and data segments. The linker concatenates the text read from multiple object files into a single text segment, and similarly concatenates the data read from multiple object files into a single data segment.

The file header of an executable file contains, for each segment, the address of the memory where it should be placed at runtime. When executing an executable file, a program called the "program loader" or simply "loader" in the operating system will copy text and data from the executable file to memory according to this information.

The following figure shows an executable file and the state of the loader loading the executable file into memory.

Executable files and memory images

In the executable shown in this figure, we assumed that the file header contained the information that the text segment was to be loaded at 0x41000 and the data segment at 0x50000.

The file header also contains information about what address to start execution from. For example, if the file header indicates that execution should start at 0x41040, the loader will load the executable file into memory as shown in the figure above, set the stack pointer to 0x7fff_ffffffff_ffffffff, and then jump to 0x41040 to start execution of the user program. The user program will then jump to 0x41040 to start execution.

Data segment content

The content of the text segment is obviously machine language, but what does the data segment contain? The answer is that the data segment contains global variables and literal strings.

Local variables are not directly in the text segment nor in the data segment. Local variables are created dynamically by the program in the stack area, so they are not particularly present immediately after loading the executable in memory.

In the C execution model, a program can start executing its main function by simply loading the executable file almost directly into memory. Therefore, global variables need to be set with appropriate initial values just by copying them from the data segment of the executable to memory.

Due to this limitation, C does not allow initialization expressions that use function calls, such as the one below, to be used for global variables.

```
int foo = bar();
```

If you have a global variable that needs dynamic initialization like the above, someone needs to execute the above expression before executing the main function. However, C does not have an initialization mechanism that is invoked before main, so such an initialization cannot be done.

In other words, a global variable must have a value that is completed at link time and can be put into the executable as a string of bytes. Such a value is limited to the following expression

- constant expression
- Addresses of global variables and functions
- The address of a global variable or function plus a constant

It should be obvious that a constant expression such as a literal number or a string can be set directly to a text segment as a fixed value.

The address of a global variable or function is usually not determined at compile time, but usually when the linker completes the executable. Thus, definitions such as `int *x = &y;`, which initializes the value of a pointer-type global variable with the address of another global variable, are legal. The linker decides the layout of the program segment by itself, and of course knows the address where the function or global variable will be loaded, so the contents of `x` can be filled in at link time.

Also, adding a constant to the address of the label is supported by the linker function, so definitions such as `int *x = &y + 3; are` legal.

Expressions other than the above pattern cannot be written in initialization expressions. For example, the value of a global variable (rather than its address) cannot be used in an initialization expression. `ptrdiff_t x = &y - &z; which` takes the difference between the addresses of two global variables, is in principle an expression that can be used to find the value at link time, but such an operation to calculate the difference between two labels is not supported by the linker. However, since the linker does not support the operation of calculating the difference between two labels, it is not possible to write such an expression in an initialization expression. Initialization expressions for global variables are allowed only in the limited patterns described above.

An example of an expression that can be written as an initialization expression for a global variable, expressed in C, is as follows

```c
int a = 3;
char b[] = "foobar";
int *c = &a;
char *d = b + 3;
```

The corresponding assembly for each expression is as follows

```
a:
 .long 3
b:
 . . byte 0x66 // 'f' .
 . . byte 0x6f // 'o'.
 . . byte 0x6f // 'o'.
 . . byte 0x62 // 'b' .
 . . byte 0x61 // 'a'.
 . . byte 0x72 // 'r'.
 . . byte 0     // '\0'
c:
 .quad a
d:
 .quad b + 3
```

Consecutive `.bytes` can also be written using the `.ascii` notation, such as `.ascii "foobar\0"`.

### Column: Dynamic Initialization of Global Variables

In C, the contents of global variables must be statically determined, but in C++, global variables can be initialized using arbitrary expressions. In other words, in C++, the global variable initialization expression is executed before the main function is called. It works through the following mechanism.

- C++ compiler compiles global variable initialization expressions into a function and outputs the function pointer to a special section called `.init_array`
- The linker concatenates the `.init_array` sections of multiple input files and outputs them in an `.init_array` segment (thus the `.init_array` segment will contain an array of function pointers)
- The loader first executes the function pointers in the `.init_array` segment in order before transferring control to `main`.

In this way, dynamic initialization of global variables is made possible by the joint efforts of the compiler, linker, and program loader.

C could support dynamic initialization of global variables if it were to use the same mechanism as C++, but such a feature has been deliberately left out of the C language specification.

The design choice of the C language specification places more restrictions on the program writer, but allows the program executor to fully satisfy the language specification even in a poor loader or loader-less environment (e.g., code that runs directly from ROM when the computer starts). Therefore, this is a matter of division, not of which is better.

## Grammar of initialization expressions

At first glance, an initialization expression looks like a simple assignment expression, but in reality, initialization expressions and assignment expressions are grammatically quite different, and there are some special ways of writing that are allowed only for initialization expressions. There are some special ways of writing that are allowed only for initialization expressions. Let's understand these special ways of writing here.

First of all, initialization expressions can be used to initialize arrays. For example, the following expression initializes `x` so that `x[0]`, `x[1]`, and `x[2]` `are` 0, 1, and 2, respectively.

```
int x[3] = {0, 1, 2};
```

If an initialization expression is given, the length of the array can be omitted because the length of the array can be found by looking at the number of elements on the right side. For example, the above expression and the one below have the same meaning.

```
int x[] = {0, 1, 2};
```

If the length of the array is given explicitly and the initialization expression is given only in part, the remaining elements must be initialized to zero. Therefore, the following two expressions have the same meaning.

```
int x[5] = {1, 2, 3, 0, 0};
int x[5] = {1, 2, 3};
```

Also, as a special syntax only for `char` array initialization expressions, the following way of writing is allowed, using a literal string as the initialization expression.

```
char msg[] = "foo";
```

The above equation is equivalent to the following equation

```
char msg[4] = {'f', 'o', 'o', '\0'};
```

## Initialization expressions for global variables

Initialization expressions for global variables need to be calculated at compile time. The result of the calculation can either be a simple byte string or a pointer to a function or global variable. In the case of pointers, it can have one integer representing the offset of the pointer.

A global variable that is not given any initialization expression at all needs to be initialized so that all bits are zero. This is the way it is defined in the C grammar.

If the initialization formula does not result in the above calculation, please treat it as a compilation error.

## Initialization expressions for local variables

Local variable initialization expressions look the same as global variable initialization expressions, but their meanings are very different. A local variable initialization expression is an expression that is executed on the fly. Therefore, its content does not have to be determined at compile time.

Basically, a statement such as `int x = 5; will` compile as if you had written it in two separate statements, such as `int x; x = 5;`.

A statement such as `int x[] = {1, 2, foo()};` will compile just like the following statement

```
int x[3];
x[0] = 1;
x[1] = 2;
x[2] = foo();
```

The content of a local variable for which no initialization expression is given is undefined. Therefore, such a variable does not need to be initialized.

**Column: Word Size**

In x86-64, the term "word" means both 16-bit data and 64-bit data. This is a complicated situation, but there are historical circumstances that make it so.

Originally, the term "word" referred to the largest integer or address size that a computer could naturally handle, which is where the term "word" comes from in the 64-bit processor x86-64, where 64 bits is one word.

When Intel engineers extended the 8086 to 32 bits to create the 386 processor, they decided to call 32 bits a double word (double word or dword) to avoid changing the size of the "word". (double word or dword). Similarly, in x86-64, which is a 64-bit extension of the 386, we call 64-bit a quad word (qword). This consideration for compatibility resulted in two different meanings of the word.

# Step 29 onwards: [additions needed].

# Static and dynamic links

This book has so far used only a feature called static linking. Static linking is a straightforward execution model, and by focusing on that model, we were able to explain the memory image of assembly code and executable files in an easy-to-understand manner. In fact, static links are not so widely used when creating general executables. However, static linking is not widely used to create executable files. In fact, dynamic linking is widely used instead of static linking.

In this chapter, we will discuss static and dynamic links.

By default, the compiler or linker will try to output an executable file that does dynamic linking. Readers may have seen the following error if you forget to add the -static option to cc by this point (if not, try removing the -static option from the Makefile and run make).

```
$ cc -o tmp tmp.s
/usr/bin/ld: /tmp/ccaRuuub.o: relocation R_X86_64_32S against
`.data' can not be used when making a PIE object; recompile with -
fPIC
/usr/bin/ld: final link failed: Nonrepresentable section on output
```

The linker will try to link dynamically by default, but in order to do so, the compiler must output assembly code that allows it to do so. is displayed if you forget to add -static. By reading this chapter, you should be able to understand what the above error means and what you need to do to fix it.

## static link

A statically linked executable is a self-contained executable that does not require any other files at runtime. For example, a function such as printf is not a user-written function but a function in the libc standard library, but when you create a statically linked executable, the printf code is copied from libc to the executable. However, when you create a statically linked executable, the printf code is copied from libc to the executable. You don't need the libc when you run a statically linked program, because the necessary code and data in the libc have already been copied to the executable.

Now let's actually see how the following simple program hello.c can be turned into a statically linked executable.

```
#include <stdio.h>.

int main() {
 printf("Hello world!\n");
}
```

To compile and link this hello.c file and convert it into an executable file with the filename hello, enter the following command

```
$ cc -c hello.c
$ cc -o hello hello.o
```

In the above command, the first line compiles hello.c to create the object file hello.o, and the second line links it to the executable file. You can also write these two commands together as cc -o hello hello.c, but when you run the compiler that way, it will internally do the same thing as the above two commands.

In hello.c, stdio.h is included, but as we have seen so far in this book, the header file does not contain the code of the function itself. Therefore, when you are creating the hello.o file, the compiler knows about the existence of the printf function declared in stdio.h and its type, but it has no knowledge of the actual code of printf. Therefore, it is impossible for the printf code to be included in the hello.o file. It is the role of the linker to combine hello.o with the object file containing printf to complete the executable.

When the linker is invoked via cc in the second line, not only the file hello.o passed on the command line is passed to the linker, but also the path /usr/lib/x86_64-linux-gnu/libc.a to the system standard library. The printf function is included in this libc.a function. The printf function is included in the libc.a function. .a is an archive file similar to .tar or .zip. Let's take a peek at what's inside.

```
$ ar t /usr/lib/x86_64-linux-gnu/libc.a
...
printf_size.o
fprintf.o
printf.o
snprintf.o
sprintf.o
...
```

printf.c in musl

The archive file contains a

The execution model of a statically linked executable is simple. Since it is the only executable in memory at runtime, each segment of the executable can be loaded to any location in memory. If you try to load to a default address that is determined at link time, the load will not fail. This is because nothing is placed in memory before the executable is loaded. Therefore, with static linking, the addresses of all global variables and functions can be determined at link time.

Static links have the following advantages

- Simple and fast to load
- Since there are no dependent files, it works by simply copying the executable.
- Even if you have different versions of a library with slightly different behavior, the library code and data are fixed when copied by static link, so the same executable will work the same way in any environment.

Static links have the following disadvantages

- The library functions and code used by the executable will be copied, so some disk and memory will be wasted.
- When a bug in a library is fixed, existing executables will need to be relinked to reflect the change.

On the other hand, dynamically linked executables require other files such as `.so` (on Unix) or `.dll` (on Windows) at runtime. The `.so` and `.dll contain the` code for functions such as `printf` and global variables such as `errno`. Files such as `.so` and `.dll are` called dynamic libraries or simply libraries, or DSO (dynamic shared object).

# C type syntax

The syntax of C types is notorious for being unnecessarily complex, and the book "Programming LanguageC" (a.k.a. "K&R"), co-authored by C developer Dennis Ritchie, states that "the syntax of C declarations, especially those involving pointers to functions, is often criticized. [7].

As you can see, the syntax of C types is not well designed, as even the author admits implicitly, but even so, the syntax is not so difficult once you understand the rules.

This chapter explains how to read the syntax of C types. Step by step, the reader should be able to decipher complex types such as void (*x)(int) and void (*signal(int, void (*)(int))(int)) by the end of this chapter.

## Diagram showing the type

The types themselves that can be represented in C are relatively simple. In order to separate the complexity of the syntax of the types from the complexity of the types themselves, let's put the syntax to the side and just think about the types.

Complex types such as pointers and arrays can be represented by a diagram of simple types connected by arrows. For example, the following diagram shows a type diagram that represents "pointer to pointer to int".

In Japanese, it is read as "a pointer to a pointer to an int" from the end point of the arrow to the beginning point. In English, you would read it the other way around: a pointer to a pointer to an int, following the direction of the arrow.

Suppose the variable x has the type shown in the figure above. The most succinct answer to the question, "What is the type of x? The most concise answer to the question "What is the type of x?" is "It's a pointer", because the type pointed to by the first arrow is a pointer. Note that x is first of all a pointer, not a type such as int. The question, "What type is the pointer pointing to? What type is the pointer pointing to?" You will also answer, "It's a pointer. This is because the pointer type is also the one you find after following one of the arrows. Finally, the question "What type is the pointer pointing to? the answer is "int".

The following figure shows an array of pointers to int. The length of the array is 20. In the actual compiler, the length of the array is also represented as a member of the type representing the array, as shown in the figure below.

If the variable x has the type shown in the above figure, then x is an array type of length 20, the elements of the array are pointers, and the pointer points to an int.

The type of a function can also be represented by a diagram. The following figure shows the type of a function that takes two arguments, int and a pointer to int, and returns a pointer to void.

Finally, let's take a more complex example. The following figure shows a type that takes an int as an argument and a pointer to a function that returns a pointer to a function that returns an int. It is complex in words, but in the diagram it is just long and simple in structure.

If the variable x has the type shown in the above figure, then x is of pointer type, the pointer points to a function, the argument type of the function is int, and the return type is pointer type.

Inside the compiler, types are represented using the same method as in the above figure. In other words, complex types related to pointers, arrays, and functions are represented inside the compiler as data structures consisting of simple type structures connected by pointers in the same order as in the above figure. Therefore, it would not be an exaggeration to say that this diagram is the true picture of types.

## Notation for types

Although the meaning of a type is easier to understand when it is represented by a diagram as shown above, it is tedious to write a diagram every time you want to understand a type. In this section, let's consider a notation that can be written more compactly without losing the clarity of the diagram.

As long as the diagram does not contain any function types, all the boxes will be arranged in a bead pattern without any branches. Therefore, if the type is only a pointer or an array, you should be able to represent the diagram with letters by writing the name of the type in the diagram directly from left to right.

Let's consider some concrete notation. Let's use the symbol * to represent a pointer box. Also, let's make it a rule that a box representing an array of length n should be written as [n], and a box representing a built-in type such as int should be written with the name of the type. Then, the following figure can be represented by the string * * int.

Since pointers, pointers, and int appear in order from the starting point of the arrow, the notation * * int is used. Conversely, if the notation * * int is given, the above diagram can also be drawn. In other words, this text representation is a notation that allows the same information as the diagram to be written down in a compact text format.

The figure below can be represented by the string [20] * int.

For functions, we will write "func(type of argument, ...) For example, the type represented in the figure below is func(int, ∗ int) ∗ void. For example, the type represented in the figure below is func(int, * int) * void. Please make sure that the notation matches the diagram.

Finally, the type represented by the figure below is * func(int) * func() int.

The notation described so far is probably the most straightforward and simple textual representation of types. In fact, the syntax of types in the Go programming language is exactly the same as the notation described here; Go is a language developed by the same people who created C, and the syntax of types in Go has been casually improved by taking advantage of C's remarks.[8]

## How to read the type of C

In this section, we will learn how to read the types of C by combining the type notation we learned above with the type notation of C.

The type C can be decomposed from the beginning into the following four parts

1. Base Type
2. An asterisk representing a pointer
3. Identifier or nested type enclosed in parentheses
4. Parentheses representing functions and arrays

For example, `int x has a` base type of `int`, no asterisk for pointers, `x` for identifiers, and no parentheses for functions or arrays. `unsigned int *x()` has a base type of `unsigned int`, an asterisk of `* for` pointers, `x` for identifiers, and parentheses () for functions. `void **(*x)() has void as the` base type, `**` as the asterisk representing the pointer, `*x` as the nested type, and `() as the` parentheses representing the function.

## How to read non-nested types

If what follows the pointer is just an identifier, it is relatively easy to decipher the type.

Without parentheses for functions and arrays, the notation for types and their meanings are as follows. The meaning will be written in the same notation as in Go, explained above.

| C type notation | Meaning |
|---|---|
| `int x` | `int` |
| `int *x` | `* int` |
| `int **x` | `* * int` |

In the presence of function parentheses, an asterisk representing the base type and a pointer will represent the return type. An example is shown below.

| C type notation | Meaning |
|---|---|
| `int x()` | `func() int` |
| `int *x()` | `func() * int` |
| `int **x(int)` | `func(int) * * int` |

You can see that the type of the function when parsed without parentheses, i.e., the type of `int x`, `int *x`, and `int **x`, is just followed by `func(...)`. You can see that they are just followed by

Similarly, when an array has parentheses, it means "an array of ~". An example is shown below.

| C type notation | Meaning |
|---|---|
| `int x[5].` | `[5] int` |
| `int *x[5].` | `[5] * int` |
| `int **x[4][5].` | `[4] [5] * * int` |

For example, in the case of the type `int *x[5]`, the type of `x` is an array of length 5, the elements of the array are pointers, and what the pointer points to is an `int`. As in the case of a function, the type of the array without the parentheses is `[...]`. You can see that the structure of the array is that it is followed by

## How to read nested types

If a pointer is followed by parentheses instead of an identifier, the parentheses represent a nested type. If there are nested types, the types inside and outside the parentheses can be parsed separately and later combined to get the overall type.

As an example, let's try to parse the declaration `int (*x)()`.

If we interpret the type of `int (*x)()` by thinking of the entire first parenthesis as a single identifier (let's say `y` as appropriate), it will look like `int y()`, which means that the type outside the parenthesis is `func() int`. On the other hand, if you think of the type `*x` inside the parentheses, it represents the type `* ___`. The type inside the parentheses does not have a base type such as `int`, so it is not a complete type, but I decided to use the notation `___` to represent the missing base part.

By parsing the type `int (*x)()` inside and outside the parentheses separately, we got the two types `func() int` and `* ___`. The type as a whole can be created by fitting the outer type into the missing part `___` of the inner type. In this case, `* func() int` is the overall type. In other words, in the declaration `int (*x)()`, `x` is a pointer, the type pointed to by the pointer is a function, and the return value of the function is `int`.

Let's take another example: `void (*x[20])(int)` follows the same pattern as `void y(int)` if we consider the first parenthesis as an identifier, so outside the parenthesis it represents the type `func(int) void`. Inside the parentheses, `*x[20]` represents `[20] * ___`. Combining the two, we get `[20] * func(int) void`. In other words, `x` is an array of length 20, the type of its elements is a pointer, the pointer points to a function that takes `int`, and the type returned by the function is `void`.

Using the method described here, it is possible to read any complex type. As an extreme example, let's try to read the type of the Unix signal function, which is famous for being a type that is completely unknown at first glance. Here is the declaration of the `signal` function

```
void (*signal(int, void (*)(int)))(int);
```

Even these complex types can be deciphered by reading them apart. First of all, let's separate the inside and outside of the first parenthesis. If we consider the first parenthesis as an identifier and think of the type, the pattern is the same as `void y(int)`, so the outside of the parenthesis represents the type `func(int) void`.

Inside the parentheses is the type `*signal(int, void (*)(int))`. This means that the base type is none, the asterisk representing a pointer is `*`, the identifier is signal, and there is a pair of parentheses representing a function with two arguments. Thus, the rough type is func(argument1, argument2) * ___. If we think about the argument types, they are as follows

- The first argument is an int. Its type is simply int.
- The second argument is void (*)(int). Since it consists of void as the base type, * as the nested type enclosed in parentheses, and a pair of parentheses representing functions, the type is func(int) void if the first parentheses are considered as identifiers. If we consider the * in the parentheses, its type is `* ___, since there is one asterisk representing a pointer. After all, the type of the second argument is a combination of those types, `* func(int) void`.

If we put the above argument types into the argument part of func, we get the type func(int, * func(int) void) * ___. Finally, combine the type inside the parentheses with the type outside the parentheses to get the final result, func(int, * func(int) void) * func(int) void.

That is, signal is a function, and the function takes the following two arguments

1. int
2. Pointer to a function that takes an int and returns a void

The return type of signal is a pointer, and the pointer points to a function that takes an int and returns a void.

### Column: C type syntax intent

C's type syntax was designed based on the idea that it would be easier to understand if we wrote types the same way we use them. With this design principle, for example, the declaration int *x[20] means that the type of x is determined in such a way that when the expression *x[20] is written, its type is int. It is the same as solving the question "What type of x matches the type of int foo = *x[20]?

If we think of the declaration int *(*x[20])() as a problem of determining the type of x so that int foo = *(*x[20])() does not cause a type error, we can assume that x must first be an array, and that the elements of the array can be dereferenced by *, and that the pointer to the array is a function, and that the return value of the array is a function that returns a pointer because it is dereferenced by *. Since the pointer is called by a function, the destination of the pointer is a function, and since the return value is dereferenced by `*`, `it is` a function that returns a pointer.

The C type syntax seems like nonsense, but it makes sense as a design policy. However, I guess I have to say that it ended up being a bad design, though.

exercises

| C type notation | Meaning |
|---|---|
| `int x` | `int` |
| `int *x` | `* int` |
| `int x[].` | `[] int` |
| `int x()` | `func() int` |
| `int **x` | `* * int` |
| `int (*x)[].` | `* [] int` |
| `int (*x)()` | `* func() int` |
| `int *x[].` | `[] * int` |
| `int x[][]` | `[] [] int` |
| `int *x()` | `func() * int` |
| `int ***x` | `* * * int` |

| C type notation | Meaning |
|---|---|
| int (**x)[] | * * [] int |
| int (**x)() | * * func() int |
| int *(*x)[]. | * [] * int |
| int (*x)[][][] | * [] [] int |
| int *(*x)() | * func() * int |
| int **x[] | [] * * Int |
| int (*x[])[] | [] * [] int |
| int (*x[])() | [] * func() int |
| int *x[][] | [] [] * int |
| int x[][][][] | [] [] [] [] int |
| int **x() | func() * * int |
| int (*x())[] | func() * [] int |
| int (*x())() | func() * func() int |

# Conclusion.

The text of this book was written in Markdown format, using Pandoc to convert Markdown to HTML, Graphviz to create syntax tree diagrams, and draw.io to create other diagrams.

[redacted]

# Appendix 1: x86-64 Instruction Set Cheat Sheet

This chapter summarizes the features of the x86-64 instruction set used by the compiler produced in this book. The following shorthand notation is used in this chapter for brevity.

- `src`, `dst`: Two arbitrary registers of the same size
- `r8`, `r16`, `r32`, `r64`: 8-bit, 16-bit, 32-bit, and 64-bit registers, respectively
- `imm`: immediate
- `reg1:reg2`: Two registers, `reg1` and `reg2`, `are` used as the upper and lower bits, respectively, to represent large numbers that cannot fit in a single register, such as 128 bits.

## List of Integer Registers

The table below shows a list of 64-bit integer registers and the names of their aliases.

| 64 | 32 | 16 | 8 |
|---|---|---|---|
| RAX | EAX | AX | AL |
| RDI | EDI | drug information | DIL |

| 64 | 32 | 16 | 8 |
|---|---|---|---|
| RSI | ESI | SI | SIL |
| RDX | EDX | DX | DL |
| rcx | ECX | CX | CL |
| RBP | EBP | BP | BPL |
| RSP | ESP | SP | SPL |
| RBX | EBX | BX | BL |
| R8 | R8D | R8W | R8B |
| R9 | R9D | R9W | R9B |
| R10 | R10D | R10W | R10B |
| R11 | R11D | R11W | R11B |
| R12 | R12D | R12W | R12B |
| R13 | R13D | R13W | R13B |
| R14 | R14D | R14W | R14B |

| 64 | 32 | 16 | 8 |
|---|---|---|---|
| R15 | R15D | R15W | R15B |

The usage in ABI is as follows. Registers that do not need to be restored to their original values when returning from a function are marked with a ✔ mark.

| register | usage (esp. of language) | |
|---|---|---|
| RAX | Return value / number of arguments | ✔ |
| RDI | First argument | ✔ |
| RSI | Second argument | ✔ |
| RDX | Third argument | ✔ |
| rcx | 4th parameter | ✔ |
| RBP | base pointer | |
| RSP | stack indicator | |
| RBX | (None in particular.) | |
| R8 | Fifth parameter | ✔ |

| register | usage (esp. of language) | |
|---|---|---|
| R9 | 6th argument | ✓ |
| R10 | (None in particular.) | ✓ |
| R11 | (None in particular.) | ✓ |
| R12 | (None in particular.) | |
| R13 | (None in particular.) | |
| R14 | (None in particular.) | |
| R15 | (None in particular.) | |

When making a function call, the `call` instruction must be called while the RSP is a multiple of 16 (aligned to 16). Function calls that do not meet this condition are not ABI-compliant and may cause some functions to crash.

## memory access

| | |
|---|---|
| mov dst, [r64]. | Load the value into dst from the address pointed by r64. |
| mov [r64], src | Store the value of src to the address pointed to by r64 |
| push r64/imm | Reduce RSP by 8 and store r64/imm in RSP |

| | |
|---|---|
| pop r64 | Load RSP to r64 and increase RSP by 8 |

## function call

| | |
|---|---|
| call label | Push RIP to stack and jump to label |
| call r64 | Push RIP onto the stack and jump to the address of r64 |
| ret | Pop the stack and jump to that address |
| leave | Equivalent to `mov rsp, rbp` then `pop rbp` |

## conditional branch

| | |
|---|---|
| cmp reg1, reg2/immje label | If reg1 == reg2/imm, jump to label |
| cmp reg1, reg2/immjne label | If reg1 ! = jump to label if reg2/imm |
| cmp reg1, reg2/immjl label | If reg1 < reg2, jump to label (Comparison with sign) |
| cmp reg1, reg2/immjle label | If reg1 <= reg2, jump to label (Comparison with sign) |

## conditional assignment

| | |
|---|---|
| cmp reg1, reg2/immsete almovzb eax, al | RAX = (reg1 == reg2) ? 1 : 0 |
| cmp reg1, reg2/immsetne almovzb eax, al | RAX = (reg1 ! = reg2) ? 1 : 0 |
| cmp reg1, reg2/immsetl almovzb eax, al | RAX = (reg1 > reg2) ? 1 : 0<br>（Comparison with sign） |
| cmp reg1, reg2/immsetle almovzb eax, al | RAX = (reg1 >= reg2) ? 1 : 0<br>（Comparison with sign） |

## Integer and logical operations

| | |
|---|---|
| add dst, src/imm | dst = dst + src/imm |
| sub dst, src/imm | dst = dst - src/imm |
| mul src | RDX:RAX = RAX * src |
| imul dst, src | dst = dst * src |
| div r32 | EAX = EDX:EAX / r32EDX = EDX:EAX % r32 |

| | |
|---|---|
| div r64 | RAX = RDX:RAX /<br>r64RDX = RDX:RAX % r64 |
| idiv r32/r64 | Signed version of div |
| cqo | RAX is sign-extended to 128 bits and stored in RDX:RAX |
| and dst, src | dst = src & dst |
| or dst, src | dst = src \| dst |
| xor dst, src | dst = src ^ dst |
| neg dst | dst = -dst |
| not dst | dst = ~dst |
| shl dst, imm/CL | Shift dst to the left by the value of the imm or CL register (only CL can be used to specify the shift amount in the register) |
| shr dst, imm/CL | Logical right shift of dst by the value of imm or CL register<br>The upper bits that have been shifted in are cleared to zero. |
| sar dst, imm/CL | Arithmetic right shift of dst by the value of imm or CL register |

| | |
|---|---|
| | The shifted-in upper bits become the same as the sign bits of the original dst. |
| lea dst, [src]. | Calculates the address of [src], but does not perform memory access, and stores the result of the address calculation itself in dst. |
| movsb dst, r8 | Sign-extend r8 and store in dst |
| movzb dst, r8 | Store r8 in dst without sign extension |
| movsw dst, r16 | Sign-extend r16 and store in dst |
| movzw dst, r16 | Store r16 in dst without sign extension |

# Appendix 2: Version Control with Git

Git was originally developed for version control of the Linux kernel, which is a huge project with thousands of developers, so Git has a wealth of features to meet the complex workflows required. While these features are useful, they are not necessary for personal development where you are the only committer. If you're new to Git, here's a cheat sheet to get you started.

- `git add file name`

Add the newly created file to the repository

- `git commit -A`

  Commit all the changes you made to your working tree at once (an editor will be launched and you can type a commit message).

- `git reset --hard`

  Undo all changes made to the working tree since the last commit.

- `git log -p`

  See past commits.

- `git push`

  Push repositories to upstreams such as GitHub

## Workflow using Git

For those of you who are new to version control systems, let me explain a little about Git and the concept of version control systems.

Git is a tool for managing a database that contains a history of changes to files. When you clone a repository from GitHub, for example, the repository is downloaded, and then the default, up-to-date directory tree is extracted from the repository to all directories under the current directory.

The directory tree extracted from the repository is called the "working tree". You edit and compile the source files in the working tree, but the working tree itself is not a part of the repository. The working tree is like an extracted file from a zip file, and no matter how many changes you make to it, the original repository remains intact.

The changes you make to your working tree are written back to the repository in "commits," which update the database and allow you to make more changes. If you use Git, you will be changing files and committing them repeatedly as you develop.

## Points to note when committing

The commit message can be in Japanese and should be written properly. For example, a one-line message such as "Add * and / as operators, but do not handle operator precedence" is fine.

The commit unit should be as small as possible. It is not desirable to combine two or more separate functions into a single commit.

You don't need to use any of the advanced features of Git. You shouldn't need to use branches, for example.

Be sure to commit the code that adds the feature together with the code that tests the feature. Also, when you commit, please run the tests first and make sure that the existing features are not broken and the new features are working properly before committing. In other words, no matter what point in time the repository is checked out, the compilation and testing will pass. However, if you inadvertently commit something that doesn't pass the tests, there is no need to modify the git commit log to fix it. Simply fix it in the next commit.

## Internal Structure of Git

There are a lot of tricky features when you read the Git documentation, but if you build a model in your mind of how Git stores data in principle, it will be easier to understand the features. So here I will explain the internal structure of Git.

Git is a kind of file system implemented as a user program, and the structure of the Git database is very similar to that of a file system. The database structure in Git is very similar to that of a file system, except that Git uses the hash value of the file as the name, whereas a normal file system uses the file name to access the file.

In a content-addressable file system, the contents of a file are identical if they have the same name. In a content-addressable file system, the contents are identical if they have the same name, and files with different contents cannot have the same name (same hash value). This is ensured by using a cryptographically secure hash function. In this kind of file system, there is no need to name the files separately, and once the name is determined, the content of the file is also uniquely determined.

A commit is also a file inside Git. The file contains not only the commit message, but also the hash value of the file that belongs to that commit and the hash value of the previous commit.

In order to retrieve a file from the Git filesystem, you need to know the hash value of the file you want.

It may seem like a chicken-and-egg problem that you can't get the commit file if you don't know the hash value of the commit file, but in fact, the repository contains the hash value of the commit file and a list of names for it, in addition to the filesystem, so you can use that to find the commit in the repository. For example, the repository contains a commit named "master" (the history of which is expanded in the working tree by default) with the hash value `da39a3ee5e`.... For example, the repository contains information such as the hash value of the commit named "master" (history expanded in the working tree by default) is da39a3ee5e. Using this information, Git can expand the files belonging to master into the working tree.

What happens internally when you "commit" is that you add the changed files to Git's internal filesystem, then you add a commit file that contains the hash of those files and the hash of the previous commit, and finally you update your inventory with the hash of that commit file. Finally, you update your inventory with the hash value of the commit file.

For example, if you want to pretend that the last commit in master never happened (which you shouldn't), you can look at the commit file pointed to by master, get the hash of the previous commit, and overwrite master with that hash. A "branch" is just another way of saying that there are two or more commits that have a commit as their previous commit, and those two commits are listed in the inventory.

This kind of content-addressable version control system also has security advantages. The name of a commit (the hash of the commit file) contains the hashes of all the files belonging to that commit, as well as the hash of the previous commit file. The commit file before that contains the hash of the commit file even before that, so in the end the hash values of all the commits that lead to the latest commit are included in the calculation of the hash value of the latest commit. Thus, it is impossible in principle to secretly alter the contents or history of a commit without changing the hash value. It's an interesting property.

Keep this content-addressable file system in mind as you learn about Git's features. It will make things a lot easier to understand.

# Appendix 3: Creating a Development Environment Using Docker

This appendix explains how to use Docker to build and run Linux applications on macOS.

Docker is a software that provides a virtual environment for Linux. The difference is that Docker provides (only) Linux system calls directly to the virtual environment, whereas a full-featured VM emulates the PC hardware and runs a regular OS on it. Docker is often used to deploy applications to computers in the cloud.

When developing Linux applications on macOS using Docker, you can consider the following two system configurations.

1. Think of the environment inside Docker and the environment outside Docker as two completely different things, and when developing Linux applications, configure all work to be done inside Docker.
2. A configuration in which normal platform-independent development tasks, such as editing source code and manipulating git, are performed outside of Docker, and only build and test commands are executed inside Docker.

In the former configuration, the configuration is simple because it is the same as preparing a Linux machine separate from the Mac for development, but the setup is a bit complicated because you need to prepare a development environment on Linux, including an editor. On the other hand, the latter configuration does not require elaborate setup of the environment since it does not "live" inside Docker, but it is a bit cumbersome to use different environments inside and outside Docker.

You can choose either of the above two configurations, but in this manual, we will choose the latter configuration to avoid explaining the setup procedure for the Linux environment. Therefore, only the commands that we want to run in the Linux environment will be explicitly executed in Docker.

## setup procedure

In order to set up a Linux development environment using Docker, you must first download and install Docker Desktop for Mac. Then, run the following command to create a Docker image named `compilerbook`.

```
$ docker build -t compilerbook
https://www.sigbus.info/compilerbook/Dockerfile
```

A "Docker image" or "image" is a collection of all the files and settings needed for a Linux environment, and the actual running of a Docker image is called a "Docker container" or simply "container" (similar to the relationship between executables and processes).

To create a container and run a command inside it, give the image name and the command as arguments to the `docker run` command. The following is an example of running the `ls /` command inside a container of `compilerbook`.

```
$ docker run --rm compilerbook ls /
bin
root
dev
etc.
...
```

The container can be made to keep running in the background, but since our usage requires only interactive usage, we gave it the `--rm` option so that the container will also be terminated when the command is terminated. Thus, every time you enter the above command, the container will be created and destroyed.

## Building with containers

In order to run `make` inside the container and compile the source files, the source files being edited outside of Docker need to be visible to the container.

In `docker run, you can` make the path <source> of the outer environment `visible` as <dest> in `Docker by` giving options of the form `-v <source>:<dest>`.

You can also specify the current directory for executing the command with the `-w` option. Using these options, you can run `make` with the directory containing the source files as the current directory.

Let's assume that the source files are located in a subdirectory named `9cc` in your home directory. To run `make test` on that directory from within the container, issue the following command

```
$ docker run --rm -v $HOME/9cc:/9cc -w /9cc compilerbook make test
```

The build and test commands should be run as described above.

If you want to start a shell in the container and use it interactively, run `docker run` with the `-it` option as shown below.

```
$ docker run --rm -it -v $HOME/9cc:/9cc compilerbook
```

## Add a new application to the container

The Docker container created in the above steps comes with a set of development tools installed, but you may want to install additional applications. This section explains how to do that.

A Docker container is a temporary entity. Docker guarantees that the application will start up in the same fresh state every time, but when you want to make changes to the image, this property is a liability. However, when you want to make changes to the image, this property can be a problem.

If you have made changes to the container and want to write them back to the image, you need to explicitly run the `docker commit` command.

As an example, let's say you want to install the `curl` command. In this case, first create a container as follows

```
$ docker run -it compilerbook
```

Note that we have not passed the `--rm` option. Then, install `curl` using `apt` from a shell inside the container as follows, and `exit` the container with the `exit` command.

```
$ sudo apt update
$ sudo apt install -y curl
$ exit
```

Since we did not add the `--rm` option when running `docker run`, the container remains in a suspended state even after exiting from the container's shell. `docker container ls -a` command can be used to view the suspended container as follows You can use the docker container ls -a command to view the suspended containers as follows

```
$ docker container ls -a
CONTAINER ID          IMAGE                 COMMAND
CREATED               STATUS                          PORTS
NAMES
a377e570d1da          compilerbook          "/bin/bash"          7
seconds ago         Exited (0) 5 seconds ago
pedantic_noyce
```

You will see that there is a container with the ID `a377e570d1da` that runs the image `compilerbook`. `docker commit` command can be used to write the container back to the image.

```
$ docker commit a377e570d1da compilerbook
```

You can make changes to the image by following the steps above.

Suspended containers and old images do not cause any problems, as they only consume some disk space, but if you are concerned about them, you can remove them by running `docker system prune`.

[redacted]

# reference data

- Compiler Explorer: A handy online compiler
- N1570 [PDF]: Final draft of the C11 language specification (identical in content to the official specification document)
- N1570: HTML version of the C11 language specification
- X4J11/86-196 [PDF]: Description of macro expansion algorithm for C preprocessor.
- Rationale for American National Standard for Information Systems - Programming Language - C: Text explaining what the standards committee considered when deciding on the C89 language specification.

- 8cc: C compiler by the author
- 9cc: C compiler by the author
- Developing a C Compiler from Scratch (Diary)
- An Incremental Approach to Compiler Construction [PDF]: A paper that gives an idea of incremental compiler development.
- Crafting Interpreters: An Online Book by Robert Nystrom
- A Retargetable C Compiler: Design and Implementation. David R. Hanson and Christopher W. Fraser: An Implementation Guide for the Simple C Compiler lcc.

[redacted]

# indexes

- lvalue
- make
- Makefile
- operator precedence
- reduced instruction set computer
- rvalue
- x64
- x86-64
- ε (Epsilon)
- assembler
- assemble
- instruction pointer
- epilogue
- overflow
- object file
- canary (esp. the island canary, Serinus canaria)
- cross compiler
- stack
- stack machine
- Segment
- data segment
- text segment
- talk about (something)
- token
- Noam Chomsky.
- flag register
- program counter
- program loader
- block
- prologue
- header file
- minus binary number

- link error
- register
- loader
- concrete syntax (of SGML)
- recursive descent parsing
- separate compilation
- forward declaration
- monadic operator
- right side value
- instruction set architecture
- Definition.
- declaration
- left side value (of an equation)
- abstract syntax tree
- context-free grammar
- most significant bit
- optimizing compiler
- syntax tree
- machine language
- Operator precedence
- production
- Number with sign
- unsigned number
- sign bit
- code extension
- final digit
- complex sentence
- Language lawyer
- function call convention
- non-terminal marker

1. http://www.drdobbs.com/cpp/increasing-compiler-speed-by-over-75/240158941

   DMD does memory allocation in a bit of a sneaky way. Since compilers are short-lived programs, and speed is of the essence, DMD just mallocs away, and never frees.

2. Linkers and Loaders, ISBN 978-1558604964, John R. Levine (1999), chapter 1.2

   Programmers were using libraries of subprograms even before they used assemblers. By 1947, John Mauchly, who led the ENIAC project, wrote about loading programs along with subprograms selected from a catalog of programs stored on tapes, and of the need to relocate the subprograms' code to reflect the addresses at which they were loaded. Perhaps surprisingly, these two basic linker functions, relocation and library search, appear to predate even assemblers, as Mauchly expected both the program and subprograms to be written in machine language. The relocating loader allowed the authors and users of the subprograms to write each subprogram as though it would start at location zero, and to defer the actual address binding until the subprograms were linked with a particular main program.

3. https://parisc.wiki.kernel.org/images-parisc/b/b2/Rad_11_0_32.pdf The 32-bit PA-RISC run-time architecture document, v. 1.0 for HP-UX 11.0, 2.2. 3章

   When a process is initiated by the operating system, a virtual address range is allocated for that process to be used for the call stack, and the stack pointer (GR 30) is initialized to point to the low end of this range. As procedures are called, the stack pointer is incremented to allow the called procedure frame to exist at the address below the stack pointer. When procedures are exited, the stack pointer is decremented by the same amount.

4. https://www.acsac.org/2002/papers/classic-multics.pdf

   Thirty Years Later: Lessons from the Multics Security Evaluation, "Third, stacks on the Multics processors grew in the positive direction, rather than the negative direction. This meant that if you actually accomplished a buffer overflow, you would be overwriting unused stack frames, rather than your own return pointer, making exploitation much more difficult.

5. https://en.wikipedia.org/wiki/ASCII#cite_note-Mackenzie_1980-1

There was some debate at the time whether there should be more control characters rather than the lowercase alphabet.

6. http://catb.org/~esr/jargon/html/L/language-lawyer.html

   language lawyer: n. A person, usually an experienced or senior software engineer, who is intimately familiar with many or most of the numerous restrictions and features (both useful and esoteric) applicable to one or more computer programming languages. A language lawyer is distinguished by the ability to show you the five sentences scattered through a 200-plus-page manual that together imply the answer to your question "if only you had thought to look there". Compare wizard, legal, legalese.

7. C Programming Language, 2nd Edition (ISBN 978-0131103627), 5.12, p.122

   C is sometimes castigated for the syntax of its declarations, particularly ones that involve pointers to functions.

8. Go's Declaration Syntax by Rob Pike