# Compiling to Assembly

## from Scratch

Vladimir Keleshev

# Compiling to Assembly

from Scratch

Vladimir Keleshev

Cover by Katiuska Pino

27 September, 2020

# Contents

**Preface** **6**

**1 Introduction** **7**
1.1 Structure of the book . . . . . 8
1.2 Why ARM? . . . . . 9
1.3 Why TypeScript? . . . . . 10
1.4 How to read this book . . . . . 10

**2 TypeScript Basics** **12**

**I Baseline Compiler** **16**

**3 High-level Compiler Overview** **17**
3.1 Types of compilers . . . . . 17
3.2 Compiler passes and intermediate representations . 18

**4 Abstract Syntax Tree** **20**
4.1 Number node . . . . . 22
4.2 Identifier node . . . . . 23
4.3 Operator nodes . . . . . 23
4.4 Call node . . . . . 26
4.5 Return node . . . . . 27
4.6 Block node . . . . . 27
4.7 If node . . . . . 28
4.8 Function definition node . . . . . 29
4.9 Variable declaration node . . . . . 30
4.10 Assignment node . . . . . 30
4.11 While loop node . . . . . 31
4.12 Summary . . . . . 32

**5 Introduction to Parser Combinators** **34**
5.1 Lexing, scanning, or tokenization . . . . . 35
5.2 Grammars . . . . . 35
5.3 Interface . . . . . 36
5.4 Primitive combinators . . . . . 39
5.5 Regexp combinator . . . . . 39
5.6 Constant combinator . . . . . 40
5.7 Error combinator . . . . . 40
5.8 Choice operator . . . . . 41
5.9 Repetition: zero or more . . . . . 43
5.10 Bind . . . . . 44
5.11 Non-primitive parsers . . . . . 45
5.12 And and map . . . . . 46
5.13 Maybe . . . . . 48
5.14 Parsing a string . . . . . 48

**6 First Pass: The Parser** **50**
6.1 Whitespace and comments . . . . . 50
6.2 Tokens . . . . . 51
6.3 Grammar . . . . . 53
6.4 Expression parser . . . . . 53
6.5 Call parser . . . . . 54
6.6 Atom . . . . . 55
6.7 Unary operators . . . . . 56
6.8 Infix operators . . . . . 56
6.9 Associativity . . . . . 58
6.10 Closing the loop: expression . . . . . 58
6.11 Statement . . . . . 58
6.12 Testing . . . . . 62

**7 Introduction to ARM Assembly Programming** **64**
7.1 A taste of assembly . . . . . 64
7.2 Running an assembly program . . . . . 66
7.3 Machine word . . . . . 67
7.4 Numeric notation . . . . . 69
7.5 Memory . . . . . 70
7.6 Registers . . . . . 71
7.7 The add instruction . . . . . 75
7.8 Immediate operand . . . . . 76
7.9 Signed, unsigned, two's complement . . . . . 77
7.10 Arithmetic and logic instructions . . . . . 78
7.11 Move instructions . . . . . 79

7.12 Program counter . . . 79
7.13 Branch instruction . . . 81
7.14 Branch and exchange . . . 81
7.15 Branch and link . . . 82
7.16 Intra-procedure-call scratch register . . . 82
7.17 Function call basics . . . 83
7.18 Link register . . . 84
7.19 Conditional execution and the CPSR register . . . 84
7.20 Conditional branching . . . 86
7.21 Loader . . . 86
7.22 Data and code sections . . . 86
7.23 Segmentation fault . . . 87
7.24 Data directives . . . 89
7.25 Loading data . . . 89
7.26 Load with immediate offset . . . 91
7.27 Storing data . . . 92
7.28 Stack . . . 92
7.29 Push and pop . . . 93
7.30 Stack alignment . . . 95
7.31 Arguments and return value . . . 96
7.32 Register conventions . . . 97
7.33 Frame pointer . . . 98
7.34 Function definitions . . . 100
7.35 Heap . . . 101

**8 Second Pass: Code Generation 103**
8.1 Test bench . . . 105
8.2 Main: entry point . . . 106
8.3 Assert . . . 107
8.4 Number . . . 108
8.5 Negation . . . 109
8.6 Infix operators . . . 110
8.7 Block statement . . . 112
8.8 Function calls . . . 112
8.9 If-statement . . . 115
8.10 Function definition and variable look-up . . . 118
8.11 Return . . . 125
8.12 Local variables . . . 126
8.13 Assignment . . . 129
8.14 While-loop . . . 130

**II Compiler Extensions** **132**

**9 Introduction to Part II** **133**

**10 Primitive Scalar Data Types** **135**

**11 Arrays and Heap Allocation** **137**
11.1 Array literals . . . 140
11.2 Array lookup . . . 141
11.3 Array length . . . 142
11.4 Strings . . . 143

**12 Visitor Pattern** **144**

**13 Static Type Checking and Inference** **148**
13.1 Scalars . . . 152
13.2 Operators . . . 152
13.3 Variables . . . 153
13.4 Arrays . . . 154
13.5 Functions . . . 155
13.6 If and While . . . 157
13.7 Error messages . . . 157
13.8 Soundness . . . 158

**14 Dynamic Typing** **159**
14.1 Tagging . . . 159
14.2 Pointer tag . . . 160
14.3 Integer tag . . . 161
14.4 Truthy and falsy tags . . . 161
14.5 Code generation . . . 161
14.6 Literals . . . 162
14.7 Operators . . . 163
14.8 Arrays . . . 164

**15 Garbage Collection** **168**
15.1 Cheney's algorithm . . . 168
15.2 Allocator . . . 169
15.3 Collection . . . 172
15.4 Discussion . . . 183
15.5 Generational garbage collection . . . 183

**16 Appendix A: Running ARM Programs** **185**
16.1 32-bit Linux on ARM (e.g. Raspberry Pi) . . . 185

16.2 64-bit Linux on ARM64 . . . . . 185
16.3 Linux on x86-64 using QEMU . . . . . 186
16.4 Windows on x86-64 using WSL and QEMU . . . . . 187

**17 Appendix B: GAS *v.* ARMASM Syntax** **189**
17.1 GNU Assembler Syntax . . . . . 190
17.2 Legacy ARMASM Syntax . . . . . 191

# Preface

Dear reader,

I am thrilled that you got yourself this book! Thank you for your support. Without your enthusiasm, this book would have been a bunch of unpublished drafts. I hope you have a blast reading it.

As you read the book or implement your compiler, please send me your thoughts. Your feedback is very appreciated. That's the beauty of the ebook format—I can make improvements, clarify things, even add or remove a chapter. My email is:

*vladimir@keleshev.com*

You can also join the book's community forum. I hope that the forum will be a great place to talk about your progress with the book, your questions, and a place to give feedback about it too: You will find the link on the book's *resources* page.

*http://keleshev.com/compiling-to-assembly-from-scratch/resources*

The forum requires an invitation code to sign up. The code is: `COM2ASM`.

The resources page also contain a link to the source code from this book: a complete compiler with a test suite.

Take care, and have fun!

*–Vladimir Keleshev*

# Chapter 1

# Introduction

It is not the gods who make our pots

*Ancient proverb*

Welcome, to the wonderful journey of writing your own compiler!

Having bought this book, you are probably already quite convinced that you want to understand how compilers work, and maybe even want to write one. Nevertheless, here's a list of some of the reasons to do it:

- Writing a compiler is the ultimate step in understanding how computers work and how they execute our programs.
- By writing a small compiler, you can see that they are programs, just like others, and they are not magic made by gods.
- By understanding assembly and how compilers translate your programs to it, you can better grasp the performance of the programs you write.
- It will allow you to see the trade-offs of different language features more clearly, so you are better informed when to use them and how to use them effectively.
- Learning about parsing will help you deal with unstructured data, like scraping, or dealing with a data format for which you don't have a library.
- It will also prepare you for making your own domain-specific languages, when necessary, for the tasks at hand.
- It may be a first step into the field of compiler engineering, a

lucrative and exciting job.
- And finally, it will allow you to create and experiment with a language of your making, and experience the fun and excitement of crafting your own language!

The topic of making compilers is the single most researched topic in computer science. Nothing else comes close. So there's a massive amount of useful techniques and algorithms in compiler literature. And it turns out, a lot of it is very applicable to our day-to-day programming, it's just that whatever we are working on today is not as well researched, at the moment. There's also a school of thought that, in the end, maybe all programs are compilers. Maybe we are not writing web apps, but compilers from DOM nodes to JSON and from JSON to SQL, who knows!

## 1.1 Structure of the book

The book describes the design and implementation of a compiler written in TypeScript, which compiles a small language to 32-bit ARM assembly.

The book consists of two parts.

*Part I* describes the design and development of a minimal *baseline compiler* in great detail. We call it a *baseline compiler* because it lays the foundation for developing more advanced features introduced in *Part II*. The *implementation language* of the compiler is TypeScript. But the compiler's *source* or *input* language is a *subset* (or a simplified version) of TypeScript. This subset consists of things common to any practical programming language, not specific to TypeScript: arithmetic and comparison operators, integer numbers, functions, conditional statements and loops, local variables, and assignments. We call this language the *baseline language*. It can express simple programs and functions, like this one, for example:

```
function factorial(n) {
  var result = 1;
  while (n != 1) {
    result = result * n;
    n = n - 1;
  }
  return result;
}
```

*Part II* builds upon the *baseline compiler* and describes various *compiler extensions* in lesser detail. Those extensions are often mutually exclusive (like static typing and dynamic typing), but they all use the baseline compiler as the foundation.

*Appendix A* describes how to run the ARM assembly code the compiler produces. You can skip this if you're developing your compiler on a computer which is based on an ARM processor with a 32-bit operating system like Raspberry Pi OS (formerly Raspbian). However, if you are running an x86-64 system like those from Intel and AMD, you need to see *Appendix A*.

*Appendix B* describes the differences between the two mainstream ARM assembly syntaxes: the GNU assembler (GAS) syntax, and the legacy ARMASM syntax.

## 1.2 Why ARM?

In many ways, the ARM instruction set is what makes this book possible.

Compared to Intel x86-64, the ARM instruction set is a work of art.

Intel x86-64 is the result of evolution from an 8-bit processor, to a 16-bit one, then to a 32-bit one, and finally to a 64-bit one. At each step of the evolution, it accumulated complexity and cruft. At each step, it tried to satisfy conflicting requirements.

- Intel x86-64 is based on *Complex Instruction Set Architecture* (CISC), which was initially optimized for writing assembly by hand.
- ARM, on the other hand, is based on *Reduced Instruction Set Architecture* (RISC), which is optimized for writing compilers.

*Guess which one is an easier target for a compiler?*

If this book targeted x86-64 instead of ARM, it would have been two times as long and—more likely—never written. Also, with 160 billion devices shipped, we better get used to the fact that ARM is the dominant instruction set architecture today.

In other words, ARM is a good start. After learning it, you will be better equipped for moving to x86-64 or the new ARM64.

## 1.3 Why TypeScript?

This book describes the design and development of a compiler written in TypeScript, which compiles a small language that also uses TypeScript syntax.

The compiler doesn't have to be written in TypeScript. It could be written in any language, but I had to pick. I have used a straightforward subset of TypeScript for the examples, to make it readable for anyone who knows one or more mainstream languages.

The next chapter, *TypeScript Basics*, gives you a quick overview of the language.

## 1.4 How to read this book

*Part I* is structured linearly, with each chapter building upon the previous one. However, don't feel guilty skipping chapters, if you are already familiar with a topic.

If you plan to follow along and implement the compiler described in this book (or a similar one), I recommend first to read *Part I* without writing any code. Then you can go back to the beginning and start implementing the compiler while skimming *Part I* again.

The book is also sprinkled with the following notes, titled *Explore…*:

> **Explore…**
>
> These notes contain suggestions and ways to try out things on your own. You might find them useful for practicing and building your confidence, or you might find it more fitting to have a minimal working compiler first, and only then optionally come back to these.

You might also see some notes titled *Well, actually…*:

> **Well, actually…**
>
> These contain some pedantic notes which are beside the point, but the book would be incomplete without them.

We will also use *code folding* in the code snippets. We will use the ellipsis (…) to denote that some code in the snippet was omitted,

usually because it was already shown before. Like this:

```
function factorial(n) {
  var result = 1;
  while (n != 1) {…}  // <- See here
  return result;
}
```

*Part II* is structured in mostly independent sections. Feel free to reach just for the parts you are interested in. No need to read both about *static typing* and *dynamic typing* if you want to focus only on one of these topics.

# Chapter 2

# TypeScript Basics

This chapter gets you up-to-speed with TypeScript. Feel free to skip it if you already know it.

TypeScript is designed to be a *superset* (or an extension) to JavaScript that adds type annotations and type-checking. So, apart from the type-annotations, it is just modern-day JavaScript.

Let's get going. First, `console.log` prints a message to the console:

```
console.log("Hello, world!");
```

Strings use double or single quotes. Strings written with back-ticks are called *template literals*. They can be multiline, *and* they can be interpolated (or injected with expressions), like in a template:

```
console.log(`2 + 2 = ${2 + 2}`); // Prints: 2 + 2 = 4
```

Next, `console.assert` is a quick and portable way to test something, if you don't have a testing framework at hand:

```
console.assert(2 === 2); // Does nothing
console.assert(0 === 2); // Raises an exception
```

Triple equals `===` is strict (or exact) equality, while double equals `==` is loose equality. Similarly with `!==` and `!=`. For example, with loose equality `true` equals to `1`, but with strict equality they are not equal. We will only use strict equality in the compiler code, but in the basline language we will use `==` and `!=` operators.

To abnormally terminate a program, you can throw an exception:

```
throw Error("Error message goes here.");
```

Functions can be defined using the `function` keyword:

```
function add(a: number, b: number) {
  return a + b;
}
```

In many cases, type-annotations are optional in TypeScript. However, they are necessary for most function (and method) parameters. So far, we were describing features that are just plain JavaScript. However, here we've got a function that has a *type annotation*. The types of the parameters are *annotated* as `number`.

Another way to define a function is by writing a so-called *arrow function*, (which is also called an *anonymous function*, or a *lambda function*):

```
let add = (a, b) => a + b;
```

Next up are variables. They come in several forms in TypeScript (and JavaScript). Variables are bound with `var`, `let` (and `const`). While `var` scope is at the function (or method) level, `let` scope is delimited with braces.

Compare `var`:

```
function f() {
  var x = 1;
  {
    var x = 2;
  }
  console.log(x); // Prints 2
}
```

With `let`:

```
function f() {
  let x = 1;
  {
    let x = 2;
  }
  console.log(x); // Prints 1
}
```

We'll only be using `let` in the compiler code, however, `var` needs

mentioning because it is `var` that we will implement for the baseline language.

Although mostly optional, bindings can also be type-annotated:

```
let a: Array<number> = [1, 2, 3];
```

Here, `Array<number>` means an array of numbers.

TypeScript, like JavaScript, allows for literal regular expressions. While strings are delimited with quotes, literal regular expressions are delimited with forward slashes:

```
let result = "hello world".match(/[a-z]+/);
console.assert(result[0] === "hello");
```

Classes are a go-to way for creating custom data types in TypeScript. Here's a simple data class:

```
class Pair {
  public first: number;
  public second: number;

  constructor(first: number, second: number) {
    this.first = first;
    this.second = second;
  }
}
```

You create a new object using the `new` keyword:

```
let origin = new Pair(0, 0);
```

TypeScript has a shortcut for defining simple data classes like this, where constructor parameters are immediately assigned as instance variables:

```
class Pair {
  constructor(public first: number,
              public second: number) {}
}
```

Note how the `public` keyword is specified in the parameter list. This is 100% equivalent to the previous definition. We will use this shortcut a lot in this book.

Here's a more extensive example, with static variable `zero`, static method `origin`, and an instance method `toString`. Remember

that static variables and static methods are just global variables and global methods that are namespaced by the class name.

```
class Pair {
  static zero = 0;

  static origin() {
    return new Pair(0, 0);
  }

  constructor(public first: number,
              public second: number) {}

  toString() {
    return `(${this.first}, ${this.second})`;
  }
}
```

There's much more to TypeScript, but we will, for the most part, limit ourselves to the subset described here.

# Part I

# Baseline Compiler

# Chapter 3

# High-level Compiler Overview

A *compiler* is a program that translates another program from one language to another.

In our case, it transforms from what we call a *baseline language* to ARM assembly language.

## 3.1 Types of compilers

Our compiler will be an *ahead-of-time* (AOT) compiler. Only once the compilation is finished can the resulting program be run.

There are also *just-in-time* (JIT) compilers that compile a program as it runs.

Think of AOT compilers as translator services for foreign languages: you might send them a few papers to translate from English to Japanese, and when they are done, they send the results back. On the other hand, JIT compilers are more like simultaneous translators at a business meeting: they translate participants as they speak.

Our compiler *targets* (or produces) an *assembly language*. An *assembly language* is a textual representation of the binary *machine language* that processors execute directly. It has a straightforward

translation to such binary. Such translation is called *assembling* and is much less sophisticated than what is found in a compiler. The program that performs this translation is called an *assembler*. In most cases on ARM, one assembly instruction is translated into one 32-bit binary integer. Think of assembly language as an API for directly accessing your processor's functionality.

Some compilers target binary *machine code* directly, but this is increasingly rare. Instead, most compilers compile to assembly and then call the assembler behind the scenes.

Some compilers target *byte code* instead of assembly. Byte code is similar to assembly: it consists of similar instructions. However, these do not target a real processor, but instead an *abstract machine*, which is a processor that is implemented in software. This could be done for portability reasons, or to add security features that are not available in hardware. Often byte code, in turn, is translated to machine code by a JIT compiler.

A possible compiler target could be another programming language. We call these compilers *source-to-source* compilers. For example, the TypeScript compiler is a source-to-source compiler that targets JavaScript.

## 3.2 Compiler passes and intermediate representations

Compilers are structured into several *passes*. At the high-level, each pass is a function that takes one representation of the program and converts it to a different representation of the program. The first such representation is the source of the program. The last one is the compiled program in the target language. In between them, we have representations that are *internal* to the compiler. We call them *intermediate representations* or IR.

In the figure you can see an example of a three-pass compiler diagram.

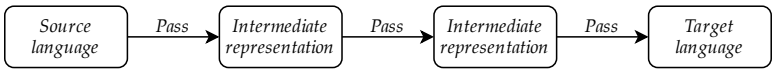


Figure 3.1: An example of a three-pass compiler

Intermediate representations of the program are data structures convenient for us to manipulate at different stages of the compiler. For one stage, we might want to use a tree-like representation. For another, we might pick a graph-like one. For some, a linear array-like representation is appropriate.

To convert from one IR to another one, each pass needs to traverse it once (or iterate through it). That's why it's called a *pass*.

The number of passes in a compiler ranges wildly, from single-pass compilers to multiple-pass compilers with dozens of passes (those are called *nano-pass* compilers).

The number of compiler passes presents a trade-off. On the one hand, we want to write many small passes that do one thing and are maintainable and testable in isolation. We also want to write more passes that do sophisticated analysis to improve the resulting programs' performance. On the other hand, we want to minimize the number of traversals to improve our compiler's performance: how fast it compiles the programs.

Our baseline compiler is a two-pass compiler. The first pass converts the source into an IR called *abstract syntax tree* or AST. This process is called *parsing*. The second pass converts from AST to assembly. It is called *emitting code* or *code generation*.

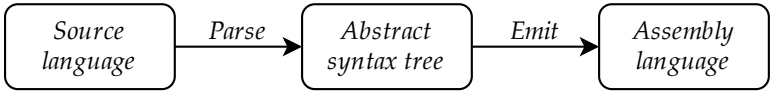


Figure 3.2: Baseline compiler structure

In *Part II* of the book, we will introduce some more passes.

Abstract syntax trees are the most common type of intermediate representations. Let's talk about them in detail.

# Chapter 4

# Abstract Syntax Tree

Abstract syntax tree, or AST, is the central concept in compilers.

AST is a data-structure. It's a *tree* that models the *syntax* of a programming language. But it *abstracts* away from the mundane details of syntax, such as the exact placement of parenthesis or semicolons.

This tree consists of *nodes*, where each node is a data object that represents a syntactic construct in the language. A `Return` node could represent a `return` statement, an `Add` node can represent a `+` operator, an identifier referring to a variable could use an `Id` node, and so on.

For example, the following line of code:

```
return n * factorial(n - 1);
```

Can have an AST like this:

```
new Return(
  new Multiply(
    new Id("n"),
    new Call("factorial", [
      new Subtract(new Id("n"), new Number(1)),
    ])))
```

In the next figure you can see a graphical representation of the same tree.

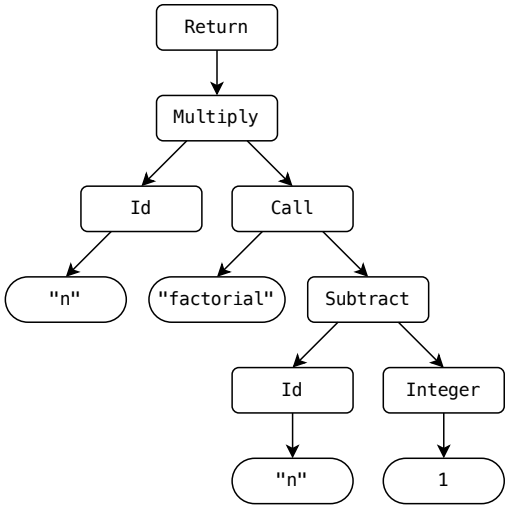


Figure 4.1: AST corresponding to `return n * factorial(n - 1)`

Why use an AST? When working with a language construct, an AST makes it convenient to operate on it: to query, construct, and reconstruct.

We design an AST so that it is convenient for us, depending on what we do with it: what kinds of transformations we want to make, which things to query or change.

For example, an AST can include source location information for error reporting. Or it can include documentation comments if our compiler needs to deal with those. Or it can include all comments and some notion of whitespace, if we want to have style-preserving transformations, like automatic refactoring.

> **Well, actually…**
>
> There are also *concrete* syntax trees, also called *parse trees*. They reflect the structure and hierarchy down to *each* input symbol. They usually have too many details that we don't care about when writing a compiler. But they are a good match for style-preserving transformations.

We will represent each kind of node in our AST as a separate class: `Return`, `Multiply`, `Id`, etc. However, at the type level, we want to

be able to refer to "any AST node". For that TypeScript gives us several tools:

- interfaces,
- abstract classes,
- union types.

Either of these works. We will use an interface:

```
interface AST {
  equals(AST): boolean;
}
```

Each type of AST node will be a class that implements the `AST` interface. It starts simple, with `equals` as the only method. We will add methods to this interface (and the classes) as needed.

The `equals` method is mostly useful for unit-testing. The implementation is quite mundane, so for the most part, we will omit it and replace its body with an ellipsis.

## 4.1 Number node

Our first node is `Number`.

```
class Number implements AST {
  constructor(public value: number) {}

  equals(other: AST) {…}
}
```

Here we used the TypeScript shortcut for quickly defining instance variables using `public` for the constructor parameter. Remember that it is equivalent to the following:

```
class Number implements AST {
  public value: number;

  constructor(value: number) {
    this.value = value;
  }

  equals(other: AST) {…}
}
```

It saves us quite some typing, which will be useful because we need to define many types of AST nodes.

We called our AST node `Number` because the data type in JavaScript and TypeScript is called `number`. However, our compiler will handle only unsigned integers.

Table 4.1: Examples of the `Number` node

| Source | AST |
|---|---|
| `0` | `new Number(0)` |
| `42` | `new Number(42)` |

## 4.2 Identifier node

Identifiers, or ids for short, refer variables in the code.

```
class Id implements AST {
  constructor(public value: string) {}

  equals(other: AST) {…}
}
```

Table 4.2: Examples of the `Id` node

| Source | AST |
|---|---|
| `x` | `new Id("x")` |
| `hello` | `new Id("hello")` |

## 4.3 Operator nodes

The next AST node is `Not`, which stands for the negation operator: "`!`".

```
class Not implements AST {
  constructor(public term: AST) {}
```

```
  equals(other: AST) {…}
}
```

Table 4.3: Examples of the `Not` node

| Source | AST |
|---|---|
| `!x` | `new Not(new Id("x"))` |
| `!42` | `new Not(new Number(42))` |

Negation is the only *prefix operator* (or *unary operator*) that we define. However, we define several *infix operators* (or *binary operators*).

Equality operator (==) and its opposite (!=).

```
class Equal implements AST {
  constructor(public left: AST, public right: AST) {}

  equals(other: AST) {…}
}

class NotEqual implements AST {
  constructor(public left: AST, public right: AST) {}

  equals(other: AST) {…}
}
```

Table 4.4: Examples of `Equal` and `NotEqual` nodes

| Source | AST |
|---|---|
| `x == y` | `new Equal(new Id("x"), new Id("y"))` |
| `10 != 25` | `new NotEqual(new Number(10), new Number(25))` |

Addition (+), subtraction (−), multiplication (∗), and division (/):

```
class Add implements AST {
  constructor(public left: AST, public right: AST) {}

  equals(other: AST) {…}
}

class Subtract implements AST {
  constructor(public left: AST, public right: AST) {}

  equals(other: AST) {…}
}

class Multiply implements AST {
  constructor(public left: AST, public right: AST) {}

  equals(other: AST) {…}
}

class Divide implements AST {
  constructor(public left: AST, public right: AST) {}

  equals(other: AST) {…}
}
```

Table 4.5: Examples of `Add` and `Subtract` nodes

| Source | AST |
|---|---|
| `x + y` | `new Add(new Id("x"), new Id("y"))` |
| `10 * 25` | `new Multiply(new Number(10), new Number(25))` |

Note that since the parameters are ASTs themselves, that means they can be arbitrarily nested.

For example, `42 + !(20 != 10)` will be:

```
new Add(new Number(42),
        new Not(new NotEqual(new Number(20),
                             new Number(10))))
```

Not all combinations of ASTs make sense. Nonetheless, this one happens to be valid in JavaScript.

## 4.4 Call node

`Call` refers to a function name (or *callee*), and an array of arguments. For example, `f(x)` becomes: `new Call("f", [new Id("x")])`.

```
class Call implements AST {
  constructor(public callee: string,
              public args: Array<AST>) {}

  equals(other: AST) {
    return other instanceof Call &&
      this.callee === other.callee &&
      this.args.length === other.args.length &&
      this.args.every((arg, i) =>
        arg.equals(other.args[i]));
  }
}
```

The language of our baseline compiler is restricted such that only named functions can be called.

We can't name the arguments array `arguments`, since this clashes with JavaScript built-in `arguments` object. So, with some reluctance, let's call it `args`.

The `Call` node is interesting because it has both a primitive string and an array of AST as its members. JavaScript doesn't have an agreed-upon protocol for equality; that's why `Call` makes an excellent example of how to implement the `equals` method in JavaScript:

- It uses `instanceof` operator to check that the other AST is also a `Call`.
- It compares the `callee` strings using the === operator.
- It uses the `.equals` method for comparing AST nodes of each argument.
- It compares array by length and checks `every` element.

Languages other than JavaScript often have more elegant ways of dealing with structural equality.

Table 4.6: Examples of the `Call` node

| Source | AST |
|---|---|
| `f(x, y)` | `new Call("f", [`<br>`  new Id("x"),`<br>`  new Id("y"),`<br>`])` |
| `factorial(n - 1)` | `new Call("factorial", [`<br>`  new Subtract(new Id("n"),`<br>`  new Number(1))])` |

## 4.5 Return node

```
class Return implements AST {
  constructor(public term: AST) {}

  equals(other: AST) {…}
}
```

Table 4.7: Examples of the `Return` node

| Source | AST |
|---|---|
| `return 0;` | `new Return(new Number(0))` |
| `return n - 1;` | `new Return(`<br>`  new Subtract(new Id("n"),`<br>`  new Number(1)))` |

## 4.6 Block node

`Block` refers to a block of code delimited with curly braces.

```
class Block implements AST {
  constructor(public statements: Array<AST>) {}

  equals(other: AST) {…}
```

```
}
```

Table 4.8: Examples of the `Block` node

| Source | AST |
|---|---|
| `{}` | `new Block([])` |
| `{`<br>`  f(x);`<br>`  return y;`<br>`}` | `new Block([`<br>`  new Call("f", [new Id("x")]),`<br>`  new Return(new Id("y")),`<br>`])` |

## 4.7 If node

The `If` node has three branches:

- `conditional` refers to the expression that is evaluated to either true or false,
- `consequence` is the branch taken in the true case, and
- `alternative` is the branch taken in the false case.

```
class If implements AST {
  constructor(public conditional: AST,
              public consequence: AST,
              public alternative: AST) {}

  equals(other: AST) { … }
}
```

This way, the following:

```
if (x == 42) {
  return x;
} else {
  return y;
}
```

Becomes:

```
new If(
  new Equal(new Id("x", new Number(42))),
  new Block([new Return(new Id("x"))]),
```

```
    new Block([new Return(new Id("y"))]),
  )
```

Curly braces are optional for if statements, thus, the following:

```
if (x == 42)
  return x;
else
  return y;
```

Becomes:

```
new If(
  new Equal(new Id("x", new Number(42))),
  new Return(new Id("x")),
  new Return(new Id("y")),
)
```

How do we represent an `if` without the `else` branch? We could have a separate node for it, or we can do a simple hack: representing `if (x) y` the same way as `if (x) y else {}`. In other words, by placing an empty `Block` as the `alternative`.

## 4.8 Function definition node

A function definition consists of a function name, an array of parameters, and the function's body.

```
class Function implements AST {
  constructor(public name: string,
              public parameters: Array<string>,
              public body: AST) {}

  equals(other: AST) { … }
}
```

Consider the following function definition.

```
function f(x, y) {
  return y;
}
```

When converted to an AST it becomes as follows.

```
new Function("f", ["x", "y"], new Block([
  new Return(new Id("y")),
```

```
]))
```

Notice that in a function definition, the parameters are strings, while in a function call, they are ASTs. This fact reflects that function calls can have nested expressions, while function definitions simply list the inbound variable names.

## 4.9 Variable declaration node

The `Var` nodes are for variable declarations. So `var x = 42;` becomes `new Var("x", new Number(42))`.

```
class Var implements AST {
  constructor(public name: string, public value: AST) {}

  equals(other: AST) {…}
}
```

Table 4.9: Examples of the `Var` node

| Source | AST |
|---|---|
| `var x = 42;` | `new Var("x", new Number(42))` |
| `var y = a + b;` | `new Var("y", new Add(new Id("a"), new Id("b")))` |

## 4.10 Assignment node

An assignment is represented with the node `Assign`.

```
class Assign implements AST {
  constructor(public name: string,
              public value: AST) {}

  equals(other: AST) {…}
}
```

Assignment differs from variable declaration in the following way: assignment changes the value of an existing variable, and does not define a new one. At least, that's the distinction that we will as-

sume. JavaScript allows assignment of a variable that is not defined yet; in such case, it will create a global variable. TypeScript, on the other hand, dissallows that.

Table 4.10: Examples of the `Assign` node

| Source | AST |
|---|---|
| `x = 42;` | `new Assign("x", new Number(42))` |
| `y = a + b;` | `new Assign("y",`<br>`  new Add(new Id("a"),`<br>`  new Id("b"))))` |

## 4.11 While loop node

The last node of our AST and the last construct in our baseline language is the while loop.

```
class While implements AST {
  constructor(public conditional: AST,
              public body: AST) {}

  equals(other: AST) {…}
}
```

Table 4.11: Examples of the `While` node

| Source | AST |
|---|---|
| `while (x)`<br>`  f();` | `new While(`<br>`  new Id("x"),`<br>`  new Call("f", []))` |
| `while (x) {`<br>`  f();`<br>`}` | `new While(new Id("x"), new Block([`<br>`  new Call("f", []),`<br>`]))` |

## 4.12 Summary

Let's look at a larger snippet converted to an AST. Remember our factorial function:

```
function factorial(n) {
  var result = 1;
  while (n != 1) {
    result = result * n;
    n = n - 1;
  }
  return result;
}
```

And here is the corresponding AST:

```
new Function("factorial", ["n"], new Block([
  new Var("result", new Number(1)),
  new While(new NotEqual(new Id("n"),
                         new Number(1)), new Block([
    new Assign("result", new Multiply(new Id("result"),
                                      new Id("n"))),
    new Assign("n", new Subtract(new Id("n"),
                                 new Number(1))),
  ])),
  new Return(new Id("result")),
]))
```

We finish this chapter with a table that summarizes all the AST constructors that we've covered, their signatures, and examples of what source code these AST nodes can represent.

In the next two chapters, you will learn about converting a program from source to an AST, or in other words, about *parsing*.

Table 4.12: Summary of AST constructor signatures with examples

| AST Constructor Signature | Example |
|---|---|
| `Number(value: number)` | `42` |
| `Id(value: string)` | `x` |
| `Not(term: AST)` | `!term` |
| `Equal(left: AST, right: AST)` | `left == right` |
| `NotEqual(left: AST, right: AST)` | `left != right` |
| `Add(left: AST, right: AST)` | `left + right` |
| `Subtract(left: AST, right: AST)` | `left - right` |
| `Multiply(left: AST, right: AST)` | `left * right` |
| `Divide(left: AST, right: AST)` | `left / right` |
| `Call(callee: string, args: Array<AST>)` | `callee(a1, a2, a3)` |
| `Return(term: AST)` | `return term;` |
| `Block(statements: Array<AST>)` | `{ s1; s2; s3; }` |
| `If(conditional: AST, consequence: AST, alternative: AST)` | `if (conditional) consequence else alternative` |
| `Function( name: string, parameters: Array<string>, body: AST, )` | `function name(p1, p2, p3) { body; }` |
| `Var(name: string, value: AST)` | `var name = value;` |
| `Assignment(name: string, value: AST)` | `name = value;` |
| `While(conditional: AST, body: AST)` | `while (conditional) body;` |

# Chapter 5

# Introduction to Parser Combinators

A parser is a function that converts some textual source into structured data. In our case, it converts a program's source into a corresponding AST.

There's a myriad of parsing techniques and approaches. Which one to choose often depends on the source language and its syntactic features.

The technique that we'll use is called *parser combinators*. We'll also use some *regular expressions* since they are so handy in JavaScript.

Parser combinators' idea is to create a small number of *primitive* parsers that could be *combined* into more complex parsers. Each primitive parser is very simple and barely does anything useful on its own, but combining those we can parse increasingly complex languages.

Parser combinators can be used to implement pretty much any parsing algorithm. They can parse immediately, or produce a data structure representing a grammar that is used later. They can even be used for code generation. In practice (and historically), many parser combinators are *scanerless* (token-less), *greedy*, *backtracking*, and with *prioritized choice*. (We'll get to what these mean in a minute). And our parsing combinators will be no exception.

Parsing combinators are my go-to technique when I need to parse

something "by hand": they are easy to implement and are powerful. After implementing a few primitive combinator functions, you get something comparable to a full-blown parser generator.

## 5.1 Lexing, scanning, or tokenization

Lexing, scanning, or tokenization are all synonyms. They refer to converting the source of a program into an intermediate representation called *token stream*. The token stream structure consists of individual "words" of our program, also called *tokens*, *lexemes*, or *lexical elements*. (A lot of this terminology originated in linguistics, which also pioneered grammars and parsing.)

A token stream is a linear representation, unlike the tree-like representation of AST.

A lexer would convert source like `x + y` into linear token stream like this:

```
[
  new TokenId("x"),
  new TokenOperator("+"),
  new TokenId("y"),
]
```

And only then a parser would convert this to an AST like `new Addition(new Id("x"), new Id("y"))`.

We've represented it using an array, but more often, it is some kind of lazy or streaming collection.

This is all to say that we won't use a lexer: our parser will be *scannerless*. The way to think about scannerless parsing is you treat each character as a token. However, we will still use the word token when we talk about parsing single logical lexical elements. We also won't go into details of the benefits and shortcomings of lexing or scannerless parsing. Here we've picked scannerless because of the implementation simplicity.

## 5.2 Grammars

When making parsers, we want to discuss what language constructs those parsers can recognize. For that, we will use *grammars*.

There are several notations to describe grammars. You might have heard about *Extended Backus-Naur Form* (EBNF). In our case, we will use *Parsing Expression Grammar* (PEG) notation. PEG notation is a good fit for us because it's designed for *scannerless*, *greedy*, *backtracking* parsers, and can express *prioritized choice*. It borrows notation from EBNF and regular expressions.

As we introduce each parser combinator, we will also show the corresponding grammar. You will see that our parser combinators are designed to mimic the grammar notation.

## 5.3 Interface

Let's say that a parser is an object with a `parse` method that takes something called `Source` and returns either something called `ParseResult` or `null`.

```
interface Parser<T> {
  parse(Source): ParseResult<T> | null;
}
```

TypeScript allows us to be explicit in that the `parse` method may return `null`, and it will also enforce this in *strict null checking* mode.

What are `Source` and `ParseResult`? Why couldn't we use `string` for the source?

When parsing, we need to keep track of the string that we parse and the location in the string where we are currently matching. Thus, `Source` is a pair consisting of:

- the string that we parse, and
- the index into that string that points to where we are parsing right now.

```
class Source {
  constructor(public string: string,
              public index: number) {}

  …
}
```

But also, `Source` allows us to avoid having tight coupling with the `string` type. For example, we can later make another implementation of `Source` that lazily reads from a file.

`Source` will be quite efficient for our use case since most `Source` objects will share the same `string` during parsing.

What about `ParseResult`? It is a simple data object that can hold some value produced by the parser and the source with the updated index position. The `value` will often be an `AST`, in our case. The `source` signifies where the parser left off, so another parser can continue from there.

```
class ParseResult<T> {
  constructor(public value: T,
              public source: Source) {}
}
```

However, parsers are allowed to return not only `ParseResult`, but also `null`. When returning `null` parser signifies that it didn't match anything. It is not an error: during a single pass, many rules will not match, but some others will.

Right now, `Source` doesn't have any operations defined. Many choices are possible. One of them is to expose a `match` method, very similar to `string.match` that takes a regular expression. The difference with `string.match` is that we need to match from a particular `source.index` position. This is possible with so-called "sticky" regular expressions.

Sticky regular expressions in JavaScript are specified with a flag `y`, like this: `/hello/y`. They are special in the way that they have a `lastIndex` property. By setting this property to some index, we can control where the regular expression will be matched. It's an odd design, but it works for us. Other programming languages have different ways to match a regular expression from a particular index.

Let's write our `match` method:

```
class Source {
  constructor(public string: string,
              public index: number) {}

  match(regexp: RegExp): (ParseResult<string> | null) {
    console.assert(regexp.sticky);
    regexp.lastIndex = this.index;
    let match = this.string.match(regexp);
    if (match) {
```

```
        let value = match[0];
        let newIndex = this.index + value.length;
        let source = new Source(this.string, newIndex);
        return new ParseResult(value, source);
      }
      return null;
    }
  }
```

First, we assert that the regexp we got is sticky; otherwise, it will not work. Then, we set `lastIndex` on the regexp to match from the source index. Then we delegate to `string.match` method to do the matching. If the match object is `null`, we return `null`: we couldn't match the regular expression. Otherwise, we got a regexp "match" object. It is an array-like object where the first item is the substring that matched the regular expression. We use that substring in two ways: First, it's the value of the `ParseResult` that we use. Second, we count the length of the matched string to advance the new `Source` index that we return as the second component of `ParseResult`.

As you can see, `Source` already *almost* satisfies our `Parser` interface.

A natural way to implement parser combinators that satisfies the `Parser` interface in TypeScript would be to create a class for each parser, with different `parser` methods. However, since `Parser` interface has only one method, and many of our parser combinators will have one-liner implementations, let's use a more lightweight approach. Instead, we'll have a single `Parser` class that takes `parse` method (as an arrow function) for a parameter:

```
class Parser<T> {
  constructor(
    public parse: (Source) => (ParseResult<T> | null)
  ) {}

  …
}
```

## 5.4 Primitive combinators

Now, let's start defining the *primitive* parser combinators. Some of them will be `Parser` instance methods; some will be static methods. We pick one or another purely based on notation: whether `f(x)` or `x.f()` is more readable for each combinator. While making this choice, we will try to mimic the corresponding grammar.

## 5.5 Regexp combinator

Our first combinator called `regexp` creates a parser that matches a regexp. It only delegates to the `source.match` method that we've just defined.

```
class Parser<T> {
  …
  static regexp(regexp: RegExp): Parser<string> {
    return new Parser(source => source.match(regexp));
  }
  …
}
```

When using this combinator, we must remember to pass "sticky" regular expressions with the `y` flag. Otherwise, the assertion that we defined earlier will remind us.

```
let hello = Parser.regexp(/hello[0-9]/y);
```

The corresponding grammar is:

```
hello <- "hello" [0-9]
```

The arrow defines a grammar rule called `hello`. As you can see, the PEG notation borrows from regular expression notation. A string in double-quotes matches literally, while `[0-9]` is a character class, a concept borrowed from regular expressions. Outside the quotes and the character class, whitespace does not matter.

Let's try to use our `hello` parser to parse a string:

```
let source = new Source("hello1 bye2", 0);
let result =
  Parser.regexp(/hello[0-9]/y).parse(source);
```

```
console.assert(result.value === "hello1");
console.assert(result.source.index === 6);
```

Here we create a new source from string `"hello1 bye2"`. We want it to parse from the beginning so we set the source index to 0. We call `parse` with the constructed source, and we get back a `ParseResult` object. Then we assert that the value being parsed is `"hello1"` and the resulting source index is advanced to 6 where the rest of the string is located: `" bye2"`.

## 5.6 Constant combinator

Next combinator might seem a little silly: it's a parser that always succeeds, returns a constant value, does not consume any input (does not advance the source). We call it `constant`. How is it good for anything? Soon, you will see. For now, let's say that it allows, when combined with other parsers, to change the return value (and the type) of a parser.

```
class Parser<T> {
  …
  static constant<U>(value: U): Parser<U> {
    return new Parser(source =>
      new ParseResult(value, source));
  }
  …
}
```

In PEG it would correspond to an empty string:

```
empty <- ""
```

The notation does not concern itself with what value is produced, only with what string is recognized.

## 5.7 Error combinator

Next is `error`: a parser that just throws an exception. The exception is not intended to be handled. It is expected to terminate the program.

```
class Parser<T> {
  …
```

```
  static error<U>(message: string): Parser<U> {
    return new Parser(source => {
      throw Error(message);
    });
  }
  …
}
```

There's no correspondence with PEG notation.

> **Explore…**
>
> A better implementation of the `error` combinator would inspect the source, convert the source index into a line-column pair, and display it together with the offending line and some context.

---

We can use these combinators with a fully qualified names, like `Parser.regexp`, or we can "import" them into the current namespace:

```
let {regexp, constant, error} = Parser;
let hello = regexp(/hello[0-9]/y);
```

From here on, we will assume that the names of all the parser combinators we define are imported, and we don't need to qualify them.

## 5.8 Choice operator

Next primitive parser combinator is the *choice operator* that we define as the `or` instance method on parsers. It allows us to select between two (or more) alternative parser choices. Unlike the previous primitive combinators, this is an instance method, not a static method. We use an instance method here purely for syntactic reasons. We would like to write `x.or(y).or(z)`, instead of `or(or(x, y), z)`, so it mimics the corresponding PEG grammar: `x / y / z`.

A parser like `left.or(right)` first tries to parse using the `left` parser. If successful, the result is returned. If not, the `right` parser is tried, and its result is returned.

```
class Parser<T> {
  …
```

```
  or(parser: Parser<T>): Parser<T> {
    return new Parser((source) => {
      let result = this.parse(source);
      if (result)
        return result;
      else
        return parser.parse(source);
    });
  }
  …
}
```

Such a choice operator is called a *prioritized choice* operator. This is in contrast with *unordered choice* operator, because it tries the alternative parsers in a (prioritized) order from left to right.

PEG notation helps us be precise about the fact that we use prioritized choice by using forward slash `/` as opposed to the unordered choice operator, which is usually denoted with a horizontal bar `|`.

Unordered choice can be handled in different ways too: a parser could explore several choices simultaneously, or the choice that could be determined unresolvable by looking at one (or $n$ characters) could be rejected outright, ahead of any parsing.

The choice operator is one of the central topics in parsing, and its choice is often the decisive factor in parser performance and recognition power.

Given a parser like `left.or(right)`, we say that it *backtracks* because it will try parser `left` and if that does not succeed *backtracks* (or returns) to the same source index and tries the `right` parser. We say that our parser has *unlimited look-ahead* because it will try to parse `left` completely, no matter how long the match is, in other words, without any *limit*. Other techniques look ahead at one (or $n$ characters) to decide which choice to take.

Theoretically, the way we implemented our choice operator is not very efficient and could be improved with caching. However, if we take some care with combining our parsers, it could be a non-problem. The key is not to combine parsers that can parse the same long prefix, but instead combine alternative parsers so that they can quickly recognize that they don't match and move to the next one.

Consider a parser constructed from the following two: one matches a hundred consecutive characters "a" followed by a single "b", and another matches a hundred consecutive characters "a" followed by a single "c". Given an input that matches the latter, it will have to scan one hundred characters *twice*:

```
regexp(/a{100}b/y).or(regexp(/a{100}c/y))
```

On the other hand, if we have one parser that parses a letter, and another one that parses a digit, we can combine them with `or` and it takes only one character to check if the first parser matches or not, before moving to the next parser:

```
let letterOrDigit =
  regexp(/[a-z]/y).or(regexp(/[0-9]/y));
```

This could be described with the following grammar:

```
letterOrDigit <- [a-z] / [0-9]
```

## 5.9 Repetition: zero or more

The next primitive parser combinator is `zeroOrMore` that handles repetition. Given a parser that returns some result, it will return a new parser that returns an array of results, instead. The implementation is as follows.

```
class Parser<T> {
  …
  static zeroOrMore<U>(
    parser: Parser<U>,
  ): Parser<Array<U>> {
    return new Parser(source => {
      let results = [];
      let item;
      while (item = parser.parse(source)) {
        source = item.source;
        results.push(item.value);
      }
      return new ParseResult(results, source);
    });
  }
  …
}
```

We apply the same parser several times in a `while` loop. Each time we advance the `source` and push the resulting value into an array. As soon as one of the parses does not match and returns `null`, we return that array as the result in a new `ParseResult` object that bundles the last `source`.

For example, this parser will match zero or more letters or digits, given `letterOrDigit` that we have just defined.

```
let someLettersOrDigits = zeroOrMore(letterOrDigit);
```

This combinator function corresponds to the $*$ operator in PEG, similar to $*$ in regular expressions:

```
someLettersOrDigits -> letterOrDigit*
```

We can also inline `letterOrDigit` and rewrite this as:

```
someLettersOrDigits -> ([a-z] / [0-9])*
```

We say that our $*$ operator is *greedy*. It is similar to PEG but unlike regular expressions. A regular expression like `a*a` will match one or more letters `a`. However, `a*a` in PEG (and in our parser combinators) will never match anything: the `a*` part of the expression will "greedily" consume input as long as it can match, and without any respect for what follows it.

## 5.10 Bind

The next parser combinator is an interesting one. It is another key combinator, on par with the choice operator. As the name alludes, it *binds* a value that is being parsed to a name. By binding a value to a name, we can manipulate and construct values that our parser produces.

```
class Parser<T> {
  …
  bind<U>(
    callback: (value: T) => Parser<U>,
  ): Parser<U> {
    return new Parser((source) => {
      let result = this.parse(source);
      if (result) {
        let value = result.value;
        let source = result.source;
```

```
        return callback(value).parse(source);
      } else {
        return null;
      }
    });
  }
  …
}
```

It takes a callback, which usually will be an arrow function. The callback takes the parsed result as the parameter and returns a new parser that should continue parsing. This way, we can combine several parsers and bind their values to construct a new return value.

For example, here we are constructing a parser for comma-separated pairs of numbers, like "12,34".

```
let pair =
  regexp(/[0-9]+/y).bind((first) =>
    regexp(/,/y).bind((_) =>
      regexp(/[0-9]+/y).bind((second) =>
        constant([first, second]))));
```

We bind the first numeric parser to the `first` parameter. We ignore the result of the comma regexp (by binding it to an underscore), then bind the second numeric regexp to the `second` parameter. From those, we construct an array, and we "return" this array by constructing a constant parser.

The corresponding grammar is:

```
pair <- [0-9]+ "," [0-9]+
```

As always, the grammar ignores the value of the parser constructs; it is only concerned with which language constructs it can recognize.

## 5.11 Non-primitive parsers

In the first example of using `bind`, we have seen two patterns that keep repeating when you use `bind` in practice. That's why we will introduce them now as non-primitive parsers.

## 5.12 And and map

The first repeating pattern is using `bind`, but ignoring the value. It effectively creates a sequence of parsers. We call this non-primitive combinator `and`. It allows to sequence parsers just like `bind`, but without binding a value to a name:

```
class Parser<T> {
  …
  and<U>(parser: Parser<U>): Parser<U> {
    return this.bind((_) => parser);
  }
  …
}
```

The next pattern that we'll see a lot is binding name only to return a constant parser immediately. We call it `map`.

```
class Parser<T> {
  …
  map<U>(callback: (t: T) => U): Parser<U> {
    return this.bind((value) =>
      constant(callback(value)));
  }
  …
}
```

It is worth mentioning that this `map` method has its parallels to the `array.map` method.

---

Now, let's try to rewrite our pair parser using the new methods:

```
let pair =
  regexp(/[0-9]+/y).bind((first) =>
    regexp(/,/y).and(
      regexp(/[0-9]+/y)).map((second) =>
        [first, second]));
```

Note that the value produced by the left parser in `left.and(right)` is ignored. If you don't want to ignore it, you need to `bind` it.

Writing code in this way is not unlike writing code that involves JavaScript promises. It can be tricky sometimes, but fortunately, given a grammar, we can make the corresponding parser almost

mechanically.

Let's look at it in another way. Let's say we have in mind a grammar, like our pair grammar:

```
pair <- [0-9]+ "," [0-9]+
```

And we want to produce a parser for it. We use `let` to define the named rule, and use `and` method for all sequences (and `or` method for any alternatives):

```
let pair =
  regexp(/[0-9]+/y).and(
    regexp(/,/y).and(
      regexp(/[0-9]+/y)));
```

And now, we should think about which values we want to extract. We want to extract the two numeric values, so we replace `and` with `bind` for them:

```
let pair =
  regexp(/[0-9]+/y).bind((first) =>
    regexp(/,/y).and(
      regexp(/[0-9]+/y).bind((second) => …)));
```

And in the last bind we construct the value we want with the `constant` parser:

```
let pair =
  regexp(/[0-9]+/y).bind((first) =>
    regexp(/,/y).and(
      regexp(/[0-9]+/y).bind((second) =>
        constant([first, second]))));
```

We're done, but if we want, we can replace the second `bind` with `map` since it returns a constant parser:

```
let pair =
  regexp(/[0-9]+/y).bind((first) =>
    regexp(/,/y).and(
      regexp(/[0-9]+/y).map((second) =>
        [first, second])));
```

## 5.13 Maybe

The `maybe` combinator allows us to parse something, or return `null` optionally. We can achieve this by combining the `or` operator with `constant(null)` parser:

```
class Parser<T> {
  …
  static maybe<U>(
    parser: Parser<U>,
  ): Parser<U | null> {
    return parser.or(constant(null));
  }
  …
}
```

Here we are optionally parsing a letter or a digit:

```
let maybeLetterOrDigit = maybe(letterOrDigit);
```

This combinator corresponds to the `?` operator in PEG, also similar to regular expression notation:

```
maybeLetterOrDigit <- letterOrDigit?
```

Like in PEG, and unlike regular expression, it is also *greedy*.

Often it is useful to have a different default rather than `null`, for example, an empty array. In those cases we can simply say `parser.or(constant([]))`.

## 5.14 Parsing a string

Although each parser has a `parse` method, this method is more convenient for composing parsers rather than using them in practice. So let's define a helper method that's a little different, `parseStringToCompletion`:

```
class Parser<T> {
  …
  parseStringToCompletion(string: string): T {
    let source = new Source(string, 0);

    let result = this.parse(source);
    if (!result)
```

```
      throw Error("Parse error at index 0");

    let index = result.source.index;
    if (index != result.source.string.length)
      throw Error("Parse error at index " + index);

    return result.value;
  }
}
```

As the name suggests, instead of taking a `Source` object, this method takes a string. It makes it more convenient for testing and using the resulting parsers. This method creates a `Source` for you and calls the `parse` method. However, unlike the `parse` method, it throws an exception if the result could not be parsed, and in case the parsing did not consume all of the input string. It also unpacks the `ParseResult` to give us only the resulting value.

---

Now that we have learned how to construct parsers let's make the parser pass for our baseline compiler!

# Chapter 6

# First Pass: The Parser

Now that we've got enough parsing machinery working, we can implement the parser that produces an AST from source for our compiler.

## 6.1 Whitespace and comments

First, let's define parsers for whitespace and comments, which we collectively refer to as *ignored*. (When a parser is split into a lexer and a parser, this is usually done at the lexer-level.)

```
let whitespace = regexp(/[ \n\r\t]+/y);
let comments =
  regexp(/[/][/].*/y).or(regexp(/[/][*].*[*][/]/sy));
let ignored = zeroOrMore(whitespace.or(comments));
```

We allow both single-line (`// …`) and multi-line (`/* … */`) comments. In JavaScript regular expressions, the dot character matches any character, *except* newline. To implement multi-line comments, we need to match it as well. It is possible to alter the meaning of the dot regular expression to mean any character *including* newline by passing a "dot-all" flag `s` alongside the "sticky" `y` flag:

```
/[/][*].*[*][/]/sy
```

We use character classes with a single character like `[*]` as a readable way to escape characters that have special meaning in regular

expressions. Otherwise, they quickly start looking like a broken saw:

```
/\/\*.*\*\//sy
```

## 6.2 Tokens

Even though our parser is scanerless (or token-less), it is still useful to distinguish tokens as our syntactic building blocks. The idea is to build token-like parsers from simple character parsers, and only then build full parser on top of token-like parsers.

As customary, we will use the naming convention for tokens (both grammar rules and parsers): they will be upper-case.

Tokens provide us a higher-level view to our source than single characters. They allow us not to deal with minute details like whitespace and comments. And this is precisely how we'll use them here. We'll define a `token` parser combinator (or, more precisely, constructor) that allows us to ignore whitespace and comments around our lexemes (or tokens).

```
let token = (pattern) =>
  regexp(pattern).bind((value) =>
    ignored.and(constant(value)));
```

It takes a regular expression pattern and returns a parser that is a sequence of that pattern and an `ignored` rule that we defined previously to handle whitespace and comments. It produces the string value matched by the pattern but ignores the value of the `ignored` rule. We "pad" with `ignored` only on the right-hand-side, not around the pattern. The latter would be redundant in all but the first token, which we can handle specially. This is a common way to handle whitespace and comments in scanerless parsers.

We'll start with defining tokens for keywords in our language, like `function`, `if`, `else`, `return`, `var`, and `while`:

```
let FUNCTION = token(/function\b/y);
let IF = token(/if\b/y);
let ELSE = token(/else\b/y);
let RETURN = token(/return\b/y);
let VAR = token(/var\b/y);
let WHILE = token(/while\b/y);
```

We've used a word-break escape sequence `\b` to make sure we don't recognize `functional` as a keyword `function` followed by identifier `al`.

Then, tokens for punctuation: commas, parenthesis, etc.:

```
let COMMA = token(/[,]/y);
let SEMICOLON = token(/;/y);
let LEFT_PAREN = token(/[(]/y);
let RIGHT_PAREN = token(/[)]/y);
let LEFT_BRACE = token(/[{]/y);
let RIGHT_BRACE = token(/[}]/y);
```

Our baseline compiler will handle only integers.

```
let NUMBER =
  token(/[0-9]+/y).map((digits) =>
    new Number(parseInt(digits)));
```

We map the integer token to convert digits to a JavaScript number using `parseInt` JavaScript built-in function. Then we construct an `Number` AST node from it.

Identifiers start with a letter or an underscore and are followed by a number of letters, numbers, and underscores. In other words, they can't start with a digit.

```
let ID = token(/[a-zA-Z_][a-zA-Z0-9_]*/y);
```

In some cases, it will be useful for us to parse identifiers just for their string value, but sometimes it is more convenient to produce the AST node. That's why we create an alias `id`, which is the same as `ID`, but instead of producing a string, it produces an AST node `Id`.

```
let id = ID.map((x) => new Id(x));
```

Now, the operator tokens:

```
let NOT = token(/!/y).map((_) => Not);
let EQUAL = token(/==/y).map((_) => Equal);
let NOT_EQUAL = token(/!=/y).map((_) => NotEqual);
let PLUS = token(/[+]/y).map((_) => Add);
let MINUS = token(/[-]/y).map((_) => Subtract);
let STAR = token(/[*]/y).map((_) => Multiply);
let SLASH = token(/[/]/y).map((_) => Divide);
```

We cannot construct a full AST node out of a single operator, but it will be handy to return the class of the AST node.

## 6.3 Grammar

The grammar for our baseline compiler is split into two parts: *expressions* and *statements*. JavaScript (and TypeScript) is pretty relaxed about distinguishing the two. In general, expressions are constructs that produce a value like `1 + 2`, and statements are something that has an effect but does not produce a value, like a `while` loop.

## 6.4 Expression parser

We will start by constructing a parser for a single expression. Here is the grammar for an expression in the baseline language.

```
args <- (expression (COMMA expression)*)?
call <- ID LEFT_PAREN args RIGHT_PAREN
atom <- call / ID / NUMBER
      / LEFT_PAREN expression RIGHT_PAREN
unary <- NOT? atom
product <- unary ((STAR / SLASH) unary)*
sum <- product ((PLUS / MINUS) product)*
comparison <- sum ((EQUAL / NOT_EQUAL) sum)*
expression <- comparison
```

It consists of infix operations such as `==`, `!=`, `+`, `-`, `*`, `/`, unary negation `!x`, identifiers, integer numbers, and function calls.

The expression rule is split into several layers in order to handle operator precedence. The `expression` itself is just an alias to `comparison`. The `comparison` rule represents the operators with the lowest precedence: `==` and `!=`. The `comparison` itself is built out of `sum` in which operators have higher precedence: `+` and `-`. The `sum`, in turn, is built out of `product` rules with the highest precedence among infix operators: `*` and `/`. Those are built from `unary`, which represents unary operators, of which we have only negation: `!`. Negation has the highest precedence among all our operators. It is built upon a rule called `atom`. The `atom` rule represents the rules that are "atomic", or require no precedence because of the way they are structured. For example, `ID` and `NUMBER` are single

lexemes, so precedence doesn't apply, while `call` and parenthesized expressions have explicit delimiters, so precedence doesn't apply either. The `args` rule is extracted to simplify the `call` rule.

This process of constructing parsers with increasing precedence is sometimes compared to adding beads to a rope.

You can probably notice that the lowest-level rules, such as `atom` and `call` (via `args`) refer back to `expression`. This means that our grammar is *recursive*. This creates a slight problem for constructing a parser for this grammar. While JavaScript allows for recursive functions, it does not allow for recursive values. Other languages allow for so-called `let-rec` bindings, but not JavaScript. The way we handle this is by initially defining `expression` as an `error` parser:

```
let expression: Parser<AST> =
  Parser.error(
    "expression parser used before definition");
```

Now we are free to use it when constructing our parser. However, we must remember to change this parser in-place once we define `comparison` and before we use it:

```
expression.parse = comparison.parse;
```

## 6.5 Call parser

We need to implement `call` by first implementing `args`.

```
args <- (expression (COMMA expression)*)?
call <- ID LEFT_PAREN args RIGHT_PAREN
```

Mechanically converting the `args` to a parser gives us:

```
// args <- (expression (COMMA expression)*)?
let args =
  maybe(
    expression.and(zeroOrMore(COMMA.and(expression))))
```

Like in the case of `Call` AST node, we called it `args` instead of `arguments`, because `arguments` is a special JavaScript object, and we don't want to clash with it.

We want this parser to return something useful, like an array of AST nodes. The `maybe` combinator returns `null` if it doesn't

match, but we want an empty array, so let's replace it with `.or(constant([]))`. We need to bind the first expression, then the rest of the expressions, and then concatenate them. By doing that we end up with:

```
// args <- (expression (COMMA expression)*)?
let args: Parser<Array<AST>> =
  expression.bind((arg) =>
    zeroOrMore(COMMA.and(expression)).bind((args) =>
      constant([arg, ...args]))).or(constant([]))
```

The expression `zeroOrMore(COMMA.and(expression))` conveniently ignores the commas and produces an array of AST nodes.

Now, implementing `call` mechanically from grammar gives us:

```
// call <- ID LEFT_PAREN args RIGHT_PAREN
let call =
  ID.and(LEFT_PAREN.and(args.and(RIGHT_PAREN)))
```

Then we bind `ID` (that represents the name of the callee) and `args` to construct a `Call` node:

```
// call <- ID LEFT_PAREN args RIGHT_PAREN
let call: Parser<AST> =
  ID.bind((callee) =>
    LEFT_PAREN.and(args.bind((args) =>
      RIGHT_PAREN.and(
        constant(new Call(callee, args))))));
```

## 6.6 Atom

The next parser is `atom`. To ensure that it produces an AST, we need to use `id` rule instead of `ID`. We also bind the `expression` to extract its value.

```
// atom <- call / ID / NUMBER
//       / LEFT_PAREN expression RIGHT_PAREN
let atom: Parser<AST> =
  call.or(id).or(NUMBER).or(
    LEFT_PAREN.and(expression).bind((e) =>
      RIGHT_PAREN.and(constant(e))));
```

The order of the choices is important. If the rule started as `ID / call` instead, `call` would never match, since `call` itself begins with an `ID`.

## 6.7 Unary operators

For unary parser, we bind both `NOT?` and `atom`. If `NOT` operator is present we construct the `Not` AST node, otherwise, we produce the underlying node.

```
// unary <- NOT? atom
let unary: Parser<AST> =
  maybe(NOT).bind((not) =>
    atom.map((term) => not ? new Not(term) : term));
```

We have bound the value of the `atom` to a name `term`, which we will use as a catch-all phrase when binding expressions or statements.

## 6.8 Infix operators

Infix operators are all constructed similarly, with lower precedence rules building upon higher precedence rules.

```
product <- unary ((STAR / SLASH) unary)*
sum <- product ((PLUS / MINUS) product)*
comparison <- sum ((EQUAL / NOT_EQUAL) sum)*
```

You can probably remember that we made sure that operator rule parsers produce the class of the corresponding AST node. For example, the `EQUAL` token produces the `Equal` class. We will use that property.

We'll start with the `product` parser. Naive mechanical translation gives us:

```
// product <- unary ((STAR / SLASH) unary)*
let product =
  unary.and(zeroOrMore(STAR.or(SLASH).and(unary)));
```

Binding the first `unary` is easy. However, constructing a nested AST out of all of this is not.

We can map `(STAR / SLASH) unary` to an intermediate data structure `{operator, term}`, which is a pair constructed from `STAR /`

`SLASH` and `unary` values. This way, we get an array of operator-term pairs. We bind the `unary` and call its value `first`.

For example, if we were parsing a string `"x * y / z"`, then `first` would be `new Id("x")`, while `operatorTerms` would be an array like this:

```
[
  {operator: Multiply, term: new Id("y")},
  {operator: Divide,   term: new Id("z")},
]
```

How do we reduce those into an AST node like the following?

```
new Divide(
  new Multiply(new Id("x"), new Id("y")),
  new Id("z"))
```

Turns out that `array.reduce` is precisely the method that can accomplish this! Here's the resulting code:

```
// product <- unary ((STAR / SLASH) unary)*
let product =
  unary.bind((first) =>
    zeroOrMore(STAR.or(SLASH).bind((operator) =>
      unary.bind((term) =>
        constant({operator, term})))).map((operatorTerms) =>
          operatorTerms.reduce((left, {operator, term}) =>
            new operator(left, term), first)));
```

Quite a mouthful! Fortunately, this is the most complicated rule that we will encounter. Even better, we can extract this repeating pattern into another parser combinator and use it again. We'll call that combinator `infix`:

```
let infix = (operatorParser, termParser) =>
  termParser.bind((term) =>
    zeroOrMore(operatorParser.bind((operator) =>
      termParser.bind((term) =>
        constant({operator, term})))).map((operatorTerms) =>
          operatorTerms.reduce((left, {operator, term}) =>
            new operator(left, term), term)));
```

We changed our `product` rule into a function that takes two parameters: `operatorParser` and `termParser`. We replaced `unary` with `termParser` and `STAR.or(SLASH)` with `operatorParser`. And now

we've got a reusable combinator, which we can use not only for `product` but also for `sum` and `comparison`.

```
// product <- unary ((STAR / SLASH) unary)*
let product = infix(STAR.or(SLASH), unary);

// sum <- product ((PLUS / MINUS) product)*
let sum = infix(PLUS.or(MINUS), product);

// comparison <- sum ((EQUAL / NOT_EQUAL) sum)*
let comparison = infix(EQUAL.or(NOT_EQUAL), sum);
```

## 6.9 Associativity

When we parse `"x * y / z"` we want it to be interpreted as `"(x * y) / z"` and produce the corresponding AST:

```
new Divide(
  new Multiply(new Id("x"), new Id("y")),
  new Id("z"))
```

Thus we say that these operators are left-associative. At the moment we don't have any right-associative operators, but if we did, we would use the same grammar rules, but we would have to use `array.reduceBack` instead of `array.reduce` to construct the AST, with some adjustments.

## 6.10 Closing the loop: expression

Now that we have defined `comparison`, we can finally define `expression` which had only a dummy implementation so far. We do that by overwriting the `parse` method of `expression`:

```
// expression <- comparison
expression.parse = comparison.parse;
```

## 6.11 Statement

Next up is parsing statements. Here's the grammar:

```
returnStatement <- RETURN expression SEMICOLON
expressionStatement <- expression SEMICOLON
```

```
ifStatement <-
  IF LEFT_PAREN expression RIGHT_PAREN
  statement ELSE statement
whileStatement <-
  WHILE LEFT_PAREN expression RIGHT_PAREN statement
varStatement <- VAR ID ASSIGN expression SEMICOLON
assignmentStatement <- ID ASSIGN EXPRESSION SEMICOLON
blockStatement <- LEFT_BRACE statement* RIGHT_BRACE
parameters <- (ID (COMMA ID)*)?
functionStatement <-
  FUNCTION ID LEFT_PAREN parameters RIGHT_PAREN
  blockStatement
statement <- returnStatement
           / ifStatement
           / whileStatement
           / varStatement
           / assignmentStatemnt
           / blockStatement
           / functionStatement
           / expressionStatement
```

The high-level structure is very similar, as in the case of expressions. It consists of several rules, with `statement` defined recursively. So we start with a dummy implementation:

```
let statement: Parser<AST> =
  Parser.error(
    "statement parser used before definition");
```

We start with a `returnStatment` that produces the `Return` AST node:

```
// returnStatement <- RETURN expression SEMICOLON
let returnStatement: Parser<AST> =
  RETURN.and(expression).bind((term) =>
    SEMICOLON.and(constant(new Return(term))));
```

The `expressionStatement` is an `expression` delimited with a semicolon:

```
// expressionStatement <- expression SEMICOLON
let expressionStatement: Parser<AST> =
  expression.bind((term) =>
    SEMICOLON.and(constant(term)));
```

The `ifStatement` produces the `If` node:

```
// ifStatement <-
//   IF LEFT_PAREN expression RIGHT_PAREN statement ELSE statement
let ifStatement: Parser<AST> =
  IF.and(LEFT_PAREN).and(expression).bind((conditional) =>
    RIGHT_PAREN.and(statement).bind((consequence) =>
      ELSE.and(statement).bind((alternative) =>
        constant(new If(conditional, consequence, alternative)))));
```

The `whileStatment` is syntactically similar to the `ifStatement`:

```
// whileStatement <-
//   WHILE LEFT_PAREN expression RIGHT_PAREN statement
let whileStatement: Parser<AST> =
  WHILE.and(LEFT_PAREN).and(expression).bind((conditional) =>
    RIGHT_PAREN.and(statement).bind((body) =>
      constant(new While(conditional, body))));
```

The `varStatment` and the `assigmentStatement` are similar as well:

```
// varStatement <-
//   VAR ID ASSIGN expression SEMICOLON
let varStatement: Parser<AST> =
  VAR.and(ID).bind((name) =>
    ASSIGN.and(expression).bind((value) =>
      SEMICOLON.and(constant(new Var(name, value)))));

// assignmentStatement <- ID ASSIGN EXPRESSION SEMICOLON
let assignmentStatement: Parser<AST> =
  ID.bind((name) =>
    ASSIGN.and(expression).bind((value) =>
      SEMICOLON.and(constant(new Assign(name, value)))));
```

Technically speaking, in JavaScript, the assignment is an expression, but for simplicity, we defined it as a statement. That's also how it's used most of the time.

The block statement is a list of statements delimited with braces. It produces a `Block` node:

```
// blockStatement <- LEFT_BRACE statement* RIGHT_BRACE
let blockStatement: Parser<AST> =
  LEFT_BRACE.and(zeroOrMore(statement)).bind((statements) =>
    RIGHT_BRACE.and(constant(new Block(statements))));
```

Function parameters are very similar to `args` that we defined previously (in terms of parser structure), but they produce an `Array<string>` instead of `Array<AST>`:

```
// parameters <- (ID (COMMA ID)*)?
let parameters: Parser<Array<string>> =
  ID.bind((param) =>
    zeroOrMore(COMMA.and(ID)).bind((params) =>
      constant([param, ...params]))).or(constant([]))
```

The function definition (which we also refer to as a statement) builds upon `parameters` and `blockStatement` to produce a `Function` node:

```
// functionStatement <-
//   FUNCTION ID LEFT_PAREN parameters RIGHT_PAREN
//   blockStatement
let functionStatement: Parser<AST> =
  FUNCTION.and(ID).bind((name) =>
    LEFT_PAREN.and(parameters).bind((parameters) =>
      RIGHT_PAREN.and(blockStatement).bind((block) =>
        constant(
          new Function(name, parameters, block)))));
```

We define `statement` as one of the above statements, and we close the loop by modifying the original `statement.parser`.

```
// statement <- returnStatement
//            / ifStatement
//            / whileStatement
//            / varStatement
//            / assignmentStatement
//            / blockStatement
//            / functionStatement
//            / expressionStatement
let statementParser: Parser<AST> =
  returnStatement
    .or(functionStatement)
    .or(ifStatement)
    .or(whileStatement)
    .or(varStatement)
    .or(assignmentStatement)
    .or(blockStatement)
    .or(expressionStatement);
```

```
statement.parse = statementParser.parse;
```

And since our language should accept more than one statement, we need one final touch to complete our parser:

```
let parser: Parser<AST> =
  ignored.and(zeroOrMore(statement)).map((statements) =>
    new Block(statements));
```

It starts with allowing the "ignored" whitespace and comments (which need to be explicitly ignored before the first logical token) and follows by zero or more statements that we convert to a `Block` node.

## 6.12 Testing

It is essential to test the parser at each step of the way. The examples from the chapter *Abstract Syntax Tree* provide a good source of unit tests. And here's an example of a possible integration test:

```
let source = `
  function factorial(n) {
    var result = 1;
    while (n != 1) {
      result = result * n;
      n = n - 1;
    }
    return result;
  }
`;

let expected = new Block([
  new Function("factorial", ["n"], new Block([
    new Var("result", new Number(1)),
    new While(new NotEqual(new Id("n"),
                           new Number(1)), new Block([
      new Assign("result", new Multiply(new Id("result"),
                                        new Id("n"))),
      new Assign("n", new Subtract(new Id("n"),
                                   new Number(1))),
    ])),
    new Return(new Id("result")),
```

```
  ])),
]);

let result = parser.parseStringToCompletion(source);

console.assert(result.equals(expected));
```

---

The parser is now complete, and so is the first pass of our compiler.

# Chapter 7

# Introduction to ARM Assembly Programming

By the end of this chapter, you will learn enough ARM assembly programming to implement the rest of the compiler.

We will be using the GNU Compiler Collection (GCC) toolchain, most notably, the GNU Assembler (GAS). For differences with ARMASM assembler, see *Appendix B*.

First, we'll take a bird-eye view of a simple hello-world program to get a taste of assembly programming. After this rough initial overview, we will dive into details.

## 7.1 A taste of assembly

This is not one of these *how to draw an owl* tutorials. I will assume that you have never done any assembly programming and walk your way through it. However, in the beginning, I wanted to start with a small, complete program to get a taste of assembly programming. After that, we'll cover each part in much more detail.

So here it is, our first program:

```
/* Hello-world program.
   Print "Hello, assembly!" and exit with code 42. */
```

```
.data
  hello:
    .string "Hello, assembly!"

.text
  .global main
  main:
    push {ip, lr}

    ldr r0, =hello
    bl printf

    mov r0, #41
    add r0, r0, #1  // Increment

    pop {ip, lr}
    bx lr
```

What this program does is it prints `Hello, assembly!` to the console and exits with error code `42`.

Now, let's discuss this piece of assembly code step-by-step.

```
.data
  hello:
    .string "Hello, assembly!"
```

The program starts with a `.data` directive. Under this directive, there are definitions of our global data, potentially mutable (or read-write). There we have only one definition, a byte string defined with a `.string` directive. It has a *label* named `hello:` which stands for the memory address of this string, which we can refer to. The data section ends, and the `.text` directive starts the *code* section.

```
.text
  .global main
  main:
    push {ip, lr}
```

This section is for immutable (read, no-write) data. It is used for constants, as well as for the actual assembly instructions. The only definition in the `.text` section is a function called `main` defined with the label `main:`. It is declared "public" using the `.global` directive. The function starts with an instruction `push` that saves

some necessary registers on the stack. Then, it continues below.

```
ldr r0, =hello
bl printf
```

The function loads the address of the string that we defined earlier, by referring to the `hello` label. The instruction `ldr` *loads* the address into *register* `r0`. The instruction `bl printf` is the *call* instruction that calls the `printf` function to print the string. The register `r0` is used to pass a parameter to `printf`.

```
mov r0, #41
add r0, r0, #1
```

Next, we set up the exit code. First, we use the `mov` instruction to *move*, or copy a number `41` into register `r0`. Then the `add` instruction increments `r0` by one, resulting in `42`. There's nothing special about the exit code `42`, and we didn't have to compute it from `41`. But this taste of assembly would be incomplete without showing some basic instructions like `mov` and `add`.

```
pop {ip, lr}
bx lr
```

The `main` function ends with a return sequence. The registers that we saved, in the beginning, are now restored with the `pop` instruction, and then we return from the function with the `bx lr` instruction, assuming that the return value is in `r0`, which should be `42`.

As you probably noticed, the familiar single- and multi-line comments are supported.

## 7.2 Running an assembly program

Here's how you get this simple program running.

> **Note:**
>
> The instructions below assume that you are running the commands on an ARM-based computer (like a Raspberry Pi) with a 32-bit operating system (like Raspberry Pi OS, formerly Raspbian). If this is not the case for you, check out *Appendix A* on how to adapt these instructions to other environments.

Save the previous program into a file called `hello.s` using a text editor. Then type the following command into the console:

```
$ gcc hello.s -o hello
```

This will instruct GCC to assemble and link our program producing a `hello` executable. By default, GCC will link our assembly with a `libc` library, which provides us with basic functions such as `printf` that we used here.

You see, the operating system kernel doesn't provide such functions directly. For example, printing to the console is implemented in libraries like `libc` on top of (operating) *system calls* like `writev`. Without these basics, we would be stuck without even being able to print to the console. However, in *Part II*, we will have a quick overview of how to make *system calls* directly.

Now, you can run the program as usual:

```
$ ./hello
Hello, assembly!
```

*Oh, hello there!*

We can check the exit code by printing the `$?` shell variable.

```
$ echo $?
42
```

---

Now that we have a template to run our programs and a rough overview, we will dive deep into the details.

We will start with the basic data structure of assembly programming, the *machine word*, then followed by an overview of how memory and registers work, and finally proceed to cover the different kinds of instructions that manipulate registers and memory.

## 7.3 Machine word

ARM is a 32-bit instruction set. That means that most operations work with a 32-bit data structure called the *machine word*.

In ARM, a word consists of 32 bits. Each bit is binary `0` or `1`.

Another way to look at it, a word consists of *four* bytes, where each byte is 8 bits.

There are also half-words and double-words. The names speak for themselves. Operations on them are not as common.

Let's look at the following word.

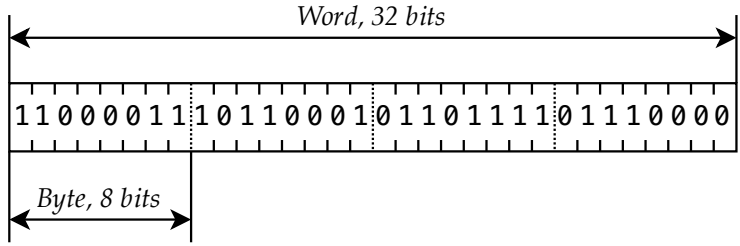


Figure 7.1: An example machine word

What does it mean? What does it stand for? Well, the processor doesn't care. It doesn't have a type system. It doesn't have any information attached to words to help us distinguish what a word stands for in isolation.

If interpreted as an unsigned integer, this word could stand for `3_283_185_520`. If interpreted as a signed integer, then it's `-1_011_781_776`. It could be a byte array of four bytes, `[195, 177, 111, 112]`. It could be an array of bits, where each bit is a single flag. Or it could be a UTF-8—encoded string `"ñop"`, where `ñ` is encoded using as two bytes `195, 177`, while `o` and `p` are encoded as single-byte ASCII characters with codes `111` and `112`. It could also be an encoding of an ARM instruction with mnemonic `rsb`. Or it could be an address pointing to some data location in memory.

It is up to us (programmers and compiler writers) to assign meaning to each word and to keep track of what they stand for.

> **Note:**
>
> Throughout this book, we'll use (non-overlapping) solid boxes of different shapes and sizes to refer to 32-bit words. Like here, we'll use dashed lines to delimit individual bytes, where it adds clarity.

## 7.4 Numeric notation

As we already did here, we'll use JavaScript notation to refer to different interpretations of data. Modern JavaScript is quite good at that. We could refer to the above word using binary *(base-2)* notation, with `0b` prefix:

```
0b11000011101100010110111101110000
```

JavaScript allows us to add underscores for readability, for example, to distinguish bit patterns of individual bytes:

```
0b11000011_10110001_01101111_01110000
```

We can use good old decimal *(base-10)* notation:

```
3_283_185_520
```

We can also use hexadecimal *(base-16)* notation, with `0x` prefix;

```
0xC3_B1_6F_70
```

How do we decide which notation to use? Why would we ever use hexadecimal?

Binary notation is very straightforward: you can see how individual bits are set, and you can visually split a word into bytes, but it is very verbose!

Decimal notation is much terser, you get a good understanding of the magnitude of the number, but it is hard to reason about the values of individual bytes and bits.

Hexadecimal notation is terse, *and* it is easy to split a word into bytes visually. Each hexadecimal digit maps to four bits, no matter the position in a number, so two hexadecimal digits always map to a byte. All you need to remember is bit patterns of the 16 hexadecimal digits:

| Hexadecimal | Binary |
|---|---|
| `0x0` | `0b0000` |
| `0x1` | `0b0001` |
| `0x2` | `0b0010` |
| `0x3` | `0b0011` |
| `0x4` | `0b0100` |
| `0x5` | `0b0101` |
| `0x5` | `0b0110` |

| Hexadecimal | Binary |
|---|---|
| `0x7` | `0b0111` |
| `0x8` | `0b1000` |
| `0x9` | `0b1001` |
| `0xA` | `0b1010` |
| `0xB` | `0b1011` |
| `0xC` | `0b1100` |
| `0xD` | `0b1101` |
| `0xE` | `0b1110` |
| `0xF` | `0b1111` |

This way we can easily translate from hexadecimal to binary and back. Take `0xC3_B1_6F_70`, as an example:

- `0xC3` is `0b1100_0011`
- `0xB1` is `0x1011_0001`
- `0x6F` is `0b0110_1111`
- `0x70` is `0b0111_0000`

Thus we can conclude that `0xC3_B1_6F_70` is the same as:

```
0b11000011_10110001_01101111_01110000
```

Can't do this with decimal notation!

## 7.5 Memory

Think of memory as a large continuous byte array. It contains our program instructions encoded as binary words. It contains the data that our program works with: data segment, code segment, stack, and heap (more on these later).

Like a byte array, you can access a single byte from memory given an index into this array. We call this index, a *memory address*. Memory addresses are 32-bit on ARM (how it all aligns, eh?).

However, not only can you access single bytes from memory, you can also access whole 32-bit words. But there is a restriction: you can only access *aligned* words. In this case, *aligned* means non-overlapping words or words which address is divisible by *four*. One word contains four bytes, and each byte has its own address, but we address words only by the address of the first byte in the word.

**Well, actually…**

Newer ARM processors support unaligned access, but not for all relevant instructions, and it incurs a performance penalty. In this book, we avoid it.

In the following figure, you can see a stretch of memory starting from address `0x00` that shows how individual bytes and their addresses map to aligned words and their addresses.

From here on, we won't need this much detail when talking about memory so that we will use a simplified (but still as precise) word-level diagrams. Like the next one that describes the same stretch of memory.

Sometimes we store a memory address in a memory word. We call that a *pointer*. In our diagrams, we will use arrows to show where a memory word is pointing. We will mostly omit the actual memory addresses in our diagrams since the exact value is not important. The important part is where it points to, not the value itself.

As we already mentioned, the memory contains our data segment, code segment, stack, and heap. However, what it does *not* contain (on most architectures, anyway) is *registers*.

## 7.6 Registers

Registers are special memory cells that are *outside* of the main memory. They are used for intermediate values, sort of like temporary variables. There's a limited number of these—usually 8, 16, or 32.

ARM has 16 main registers and a special *status* register (CPSR). The main registers are called `r0` to `r15`, but some of them have alternative names. See the next figure for more details.

First, why do we need registers? Couldn't instructions work directly with memory? They could, and there are other architectures such as accumulator-based and stack-based architectures that need only one register or no registers at all. However, ARM is a *load-store* architecture.

With *load-store* architecture, the basic workflow is as follows:

- data is loaded from memory into registers, then
- operations are performed on registers, and finally
- the data is stored back into memory.

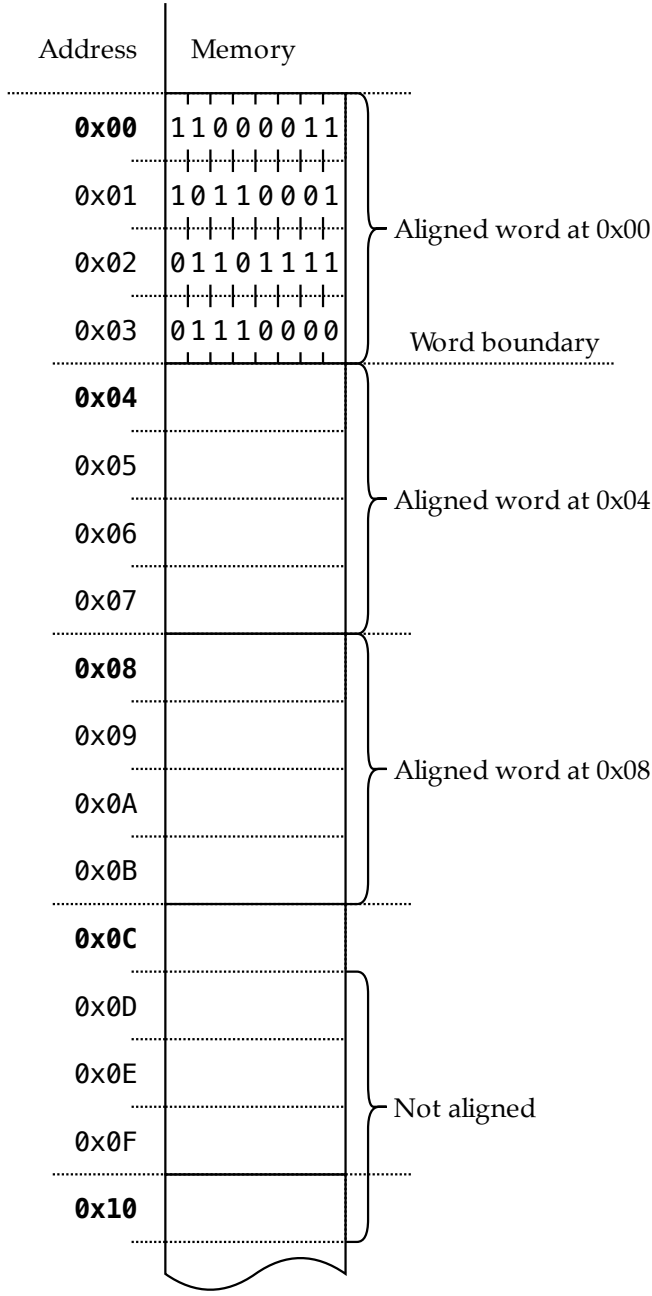


Figure 7.2: An example stretch of memory on byte-level

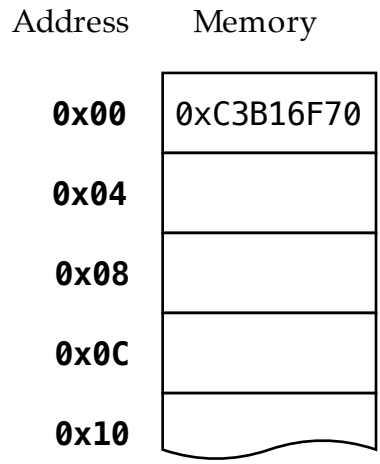


Figure 7.3: Same stretch of memory on word-level

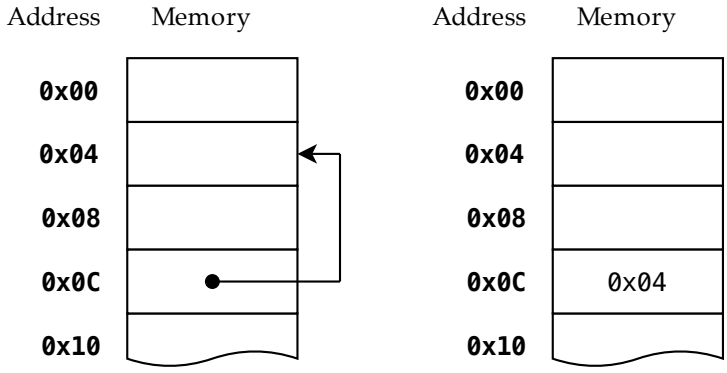


Figure 7.4: Pointer notation: left—with arrows, right—with actual values

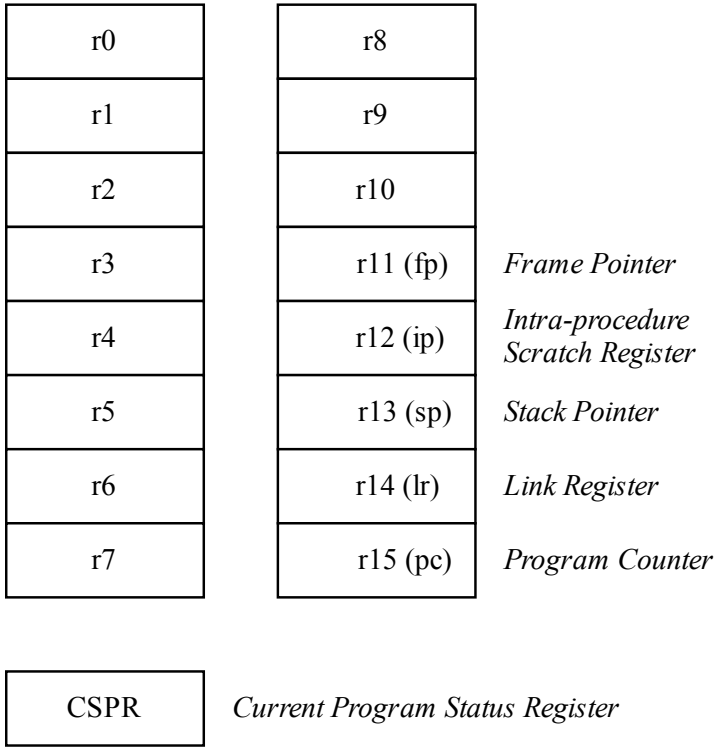


Figure 7.5: Figure about registers

It turns out this workflow is quite efficient, and most modern architectures follow it.

Most ARM registers are general-purpose and can be used for any intermediate values. We'll cover the more *special-purpose* registers like `fp`, `ip`, `sp`, `lr`, `pc`, and CPSR as we discover instructions that work with them.

---

You have maybe heard that *registers are fast*. What does that even mean? Why couldn't we use the same technology for memory?

Two reasons.

There are only 16 registers. That means you can encode a register using only 4 bits. At the same time, memory addresses are 32-bit. So you need fewer bytes (and instructions) to encode an operation on three registers, rather than an operation on three memory addresses. And a processor can decode fewer instructions faster.

Second, computer memory has several *levels* of caches, usually referred to as *L1–L3*. And even if the fastest cache uses the same technology as registers, there could still be a *cache miss*. But such a cache miss can never happen in the case of registers.

## 7.7 The add instruction

Let's get to our first instruction, `add`:

```
add r1, r2, r3      /* r1 = r2 + r3; */
```

It consists of a mnemonic name `add` as well as three register *operands*, `r1`, `r2`, and `r3`. In this case, *operand 1* is `r2`, *operand 2* is `r3`, and `r1` is the result, also called the *destination* operand. This kind of instruction is called three-operand instruction.

As a comment, we provided a pseudo-code that describes the effect of the instruction. Note that the order of operands in the instructions is the same as in the pseudo-code. ARM assembly was designed such that this is always the case.

All ARM instructions are encoded into single 32-bit words in memory. In this figure, you can see how this particular instruction is encoded into binary form. We've left the meaning of some of the bits unexplained.

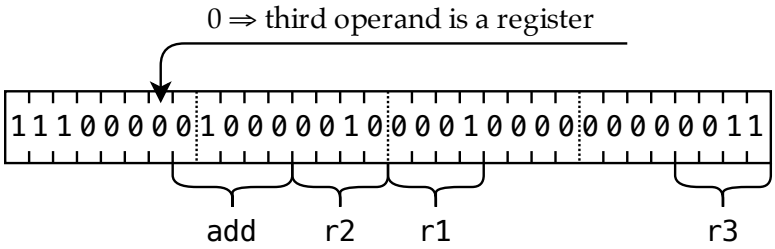


Figure 7.6: The encoding of: `add r1, r2, r3`

*Well, actually…*

Most ARM processors support several instruction sets. They have been historically called ARM, Thumb, and ARM64, but recently renamed to A32, T32, and A64.

T32 (or Thumb 2), for example, is a variable-length instruction set with both 16-bit and 32-bit instructions.

## 7.8 Immediate operand

The `add` instruction has a second form, where the *last* operand is a small number encoded directly into the instruction. It is called an *immediate* operand, and the notation uses a `#` sign:

```
add r1, r2, #64000      /* r1 = r2 + 64000; */
```

GNU Assembler allows familiar syntax for hexadecimal values with `0x` prefix and binary values with `0b` prefix. However, it doesn't allow underscores in them. So, the previous instruction can be rewritten as:

```
add r1, r2, #0xFA00      /* r1 = r2 + 0xFA00; */
```

In the following figure you can see how this instruction is encoded.

From the figure, you can see that there are 8 bits dedicated to the immediate operand, so you might conclude that it can represent any single byte value. But—wait!—we just used `64000` in our example! It turns out, there are four more bits in the instruction that encode how many *even* number of bits the immediate should be shifted. This way we can represent `0xFA`, or `0xFA0`, or `0xFA00`, or `0xFA000` and so on.

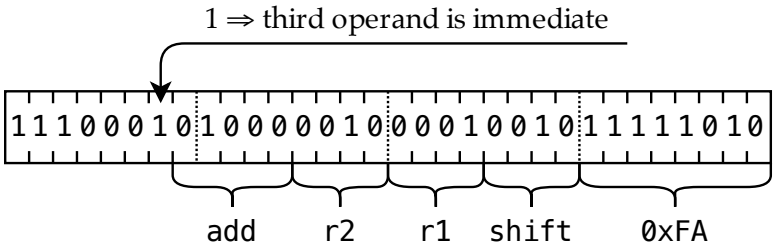


Figure 7.7: The encoding of: `add r1, r2, #0xFA00`

This is an ingenious way to encode a vast amount of interesting constants in a very tight space!

## 7.9 Signed, unsigned, two's complement

What are we adding with the `add` instruction? Unsigned integers? Signed integers?

It turns out that it works correctly both when all the operands are treated as unsigned integers, and in the case where they are all treated as signed integers. This is thanks to the signed number representation that most computers use, called *two's complement*. It was specifically designed for this trick: to use the same hardware adder for both signed and unsigned numbers.

> **Note:**
>
> Even though `add` and most ARM instructions work on 32-bit words, in this section we'll show examples using signed and unsigned 8-bit bytes, to make them more manageable.

For example, if we try to add `0b1111_1100` and `0b0000_0010` using a hardware adder, we get `0b1111_1110`, which could be interpreted as an unsigned operation $252 + 2 \Rightarrow 254$, or as a signed operation $-4 + 2 \Rightarrow -2$. In the following table, you can see how a range of binary patterns can be interpreted as an unsigned or a signed integer.

| Bit pattern | Unsigned interpretation | Signed interpretation |
|---|---|---|
| `0b0000_0000` | `0` | `0` |
| `0b0000_0001` | `1` | `1` |
| `0b0000_0010` | `2` | `2` |
| `0b0000_0011` | `3` | `3` |
| ⋮ | ⋮ | ⋮ |
| `0b0111_1101` | `125` | `125` |
| `0b0111_1110` | `126` | `126` |
| `0b0111_1111` | `127` | `127` |
| | | *— signed overflow —* |
| `0b1000_0000` | `128` | `-128` |
| `0b1000_0001` | `129` | `-127` |
| `0b1000_0010` | `130` | `-126` |
| ⋮ | ⋮ | ⋮ |
| `0b1111_1100` | `252` | `-4` |
| `0b1111_1101` | `253` | `-3` |
| `0b1111_1110` | `254` | `-2` |
| `0b1111_1111` | `255` | `-1` |
| | *— unsigned overflow —* | |
| `0b0000_0000` | `0` | `0` |
| `0b0000_0001` | `1` | `1` |
| `0b0000_0010` | `2` | `2` |
| `0b0000_0011` | `3` | `3` |
| ⋮ | ⋮ | ⋮ |

Two's complement is an elegant system, but we won't detail it here.

## 7.10 Arithmetic and logic instructions

So, we have covered our first instruction. Took a while, huh? And I have good news for you! All arithmetic and logic instructions in ARM have precisely the same three-operand form!

Here are just some of them:

| Instruction | Mnemonic | Effect |
|---|---|---|
| `add r1, r2, r3` | Add | `r1 = r2 + r3;` |
| `sub r1, r2, r3` | Subtract | `r1 = r2 - r3;` |

| Instruction | Mnemonic | Effect |
|---|---|---|
| `mul r1, r2, r3` | Multiply | `r1 = r2 * r3;` |
| `sdiv r1, r2, r3` | Signed divide[1] | `r1 = r2 / r3;` |
| `udiv r1, r2, r3` | Unsigned divide[1] | `r1 = r2 / r3;` |
| `bic r1, r2, r3` | Bitwise clear | `r1 = r2 & ~r3;` |
| `and r1, r2, r3` | And (bitwise) | `r1 = r2 & r3;` |
| `orr r1, r2, r3` | Or (bitwise) | `r1 = r2 \| r3;` |
| `eor r1, r2, r3` | Exclusive or (bitwise) | `r1 = r2 ^ r3;` |

Neat, isn't it? We have now basically covered a big chunk of the instruction set. Let's move on.

## 7.11 Move instructions

Move instructions copy a word from one register to another, or from an immediate operand to a register. An immediate operand has the same restrictions as before. There's also a "move–not" instruction that does bitwise negation.

| Instruction | Mnemonic | Effect |
|---|---|---|
| `mov r1, r2` | Move | `r1 = r2;` |
| `mvn r1, r2` | Move-not | `r1 = ~r2;` |

## 7.12 Program counter

Now we know that each instruction is encoded into a word. And that instructions are located in memory one after another. How does execution go from one instruction to another?

For that, the *program counter* is used. The program counter is the register `r15`, but more often, it is referred to by its alternative name: `pc`. On some other architectures, it is called *instruction pointer*. The program counter is a pointer that points to the currently executing

[1] Division is one of those operations that differ for signed and unsigned integers. As JavaScript doesn't have proper support for unsigned integers, we can't express the difference easily with our pseudo-code notation.

instruction. By manipulating `pc`, we can change which instruction is executing next.

> **Well, actually…**
>
> Because of instruction pipelining (which we won't cover here) program counter usually points two instructions ahead of the currently executing instruction. But, for the most part, we can safely ignore that.

For example, if register `r0` maintains some address that we want to *jump* to we can do that by moving that address into `pc` and the execution will continue from there:

```
mov pc, r0        /* pc = r0; */
```

By *jump*, we mean that execution is transferred to a different instruction, not the next one.

It is worth highlighting that *every* instruction affects the program counter. At the very least, each instruction increments program counter by four bytes (one word) so that execution can transfer to the next instruction. (Otherwise, we would always be stuck executing the same instruction). So the *effect* of each instruction that we listed so far requires `pc += 4;` to be prepended:

```
add r1, r2, r3     /* pc += 4; r1 = r2 + r3; */
```

Here's a little snippet of assembly where we increment `r0` by one (at a time), but by adding to `pc` we jump over two instructions.

```
/* This is not recommended in practice.       */
mov r0, #0       /* pc += 4; r0 = 0;          */
add r0, r0, #1   /* pc += 4; r0 = r0 + 1;     */
add r0, r0, #1   /* pc += 4; r0 = r0 + 1;     */
add pc, pc, #8   /* pc += 4; pc = pc + 8;     */
add r0, r0, #1   /* …skipped…                 */
add r0, r0, #1   /* …skipped…                 */
add r0, r0, #1   /* pc += 4; r0 = r0 + 1;     */
```

Here, we made all the changes in `pc` explicit. Note that even though `add` typically increments `pc` by 4, this is overridden if the result is written to `pc`.

From here on, we will continue omitting writing `pc += 4;` in our pseudo-code.

---

We can conclude that the program counter is the most special register of all. By changing it, we change which instruction is executed in our program next. So don't use it for storing some temporary values!

## 7.13 Branch instruction

We could jump forward and backward in code by adding to and subtracting from the program counter, but it is very cumbersome. Every time we add an instruction to our program, the offset of all the following instructions addresses.

We can use a little bit of help from the assembler. Assembler allows us to insert textual *labels* that represent particular instruction addresses. Then we can use them with *branch* instruction `b`, and the assembler will take care of calculating the offset that is necessary to apply to `pc` to land at the correct instruction.

This way, our previous snippet can be rewritten as:

```
    mov r0, #0
    add r0, r0, #1
    add r0, r0, #1
    b myLabel        /* pc = myLabel; */
    add r0, r0, #1   /* …skipped… */
    add r0, r0, #1   /* …skipped… */
  myLabel:
    add r0, r0, #1
```

Even though the `b` instruction jumps to an offset relative to `pc`, and not to an absolute address, it helps to think of labels as constants with absolute addresses stored in them. And that is how we will use them in the pseudo-code describing the effects of each instruction:

```
    b myLabel        /* pc = myLabel; */
```

## 7.14 Branch and exchange

While `b` instruction allows us to make a relative jump computed from a label, `bx`, or *branch and exchange*, it allows us to jump to an address stored in a register:

```
bx r0       /* pc = r0; */
```

How is it different from `mov pc, r0`? Not much. For our purpose, it is the same. Some would say it's a bit more readable.

However, it also allows to *exchange* instruction sets from ARM to Thumb and back. But we won't be dealing with that in this book.

## 7.15 Branch and link

*Branch and link*, or `bl` is a relative jump just like `b`. The only difference is that it saves the program counter's value into a particular register, `r14`, more often referred to as `lr`, or *link register*.

```
bl myLabel    /* lr = pc; pc = myLabel; */
```

What it effectively does is it *saves* the previous value of `pc` before overriding it (and thus, losing it).

We could achieve the same with the following two instructions:

```
mov lr, pc;
b myLabel;
```

However, this is such a common operation that it deserved its own instruction. It is common because it is used to implement function calls.

## 7.16 Intra-procedure-call scratch register

Note that, under the hood, the instructions that branch to a label (`b` and `bl`) encode a 24-bit immediate value for the relative jump. That allows us to jump forward and backward within ±32 MB of code space. But you don't have to deal with this limitation. The linker will arrange a so-called *veneer*: jump to a special place within the ±32 MB limit and then load a full 32-bit address into `pc`, if a longer jump is necessary.

To do that, it will need to generate code that loads the full address into a temporary register. For this to be predictable, a special register was designated for the role: the *intra-procedure–call scratch register*, referred to as `r12` or `ip`. This is mostly relevant for procedure (in other words, function) calls since other jumps to a label are usually relatively short. Why *intra*? Because it is used *between*

the calls: after the call is made by the caller, but before the control switches to the callee function. It is also referred to as a *scratch* register, meaning a short-lived temporary. We can still use this register for our temporary values; we just can't rely on that the value will be preserved after a call is made. So it is best-suited for short-lived temporary values.

## 7.17 Function call basics

The thing special about function calls that distinguishes them from other kinds of control-flow jumps is that function calls *return back*. When programming in a higher-level language, we don't think much about that, but at assembly level, we have to implement the whole *return back* thing ourselves.

The value that the `bl` instruction stores into `lr` is called the *return address*. When the function finishes, it can jump to it to continue execution from where it was called.

Let's implement a very primitive function, `addFourtyTwo`, that takes a single parameter and adds `42` to it. ARM calling convention (which will talk more about later) says that we should pass the first four arguments in registers `r0`–`r3` (if any), and provide a return value in `r0` (if any, again). In our case, `addFourtyTwo` takes one parameter and returns one value so that we use `r0` for both.

```
    mov r0, #0          /* r0 = 0;                      */
    bl addFourtyTwo     /* lr = pc; pc = addFourtyTwo; */
    sub r0, r0, #3      /* r0 = r0 - 3;                 */

  addFourtyTwo:
    add r0, r0, #42     /* r0 = r0 + 42;                */
    bx lr               /* pc = lr;                     */
```

The *caller* that calls `addFourtyTwo` first sets register `r0` to `0`, then uses `bl` to jump to `addFourtyTwo`, which saves the return address into `lr`. At the end of `addFourtyTwo`, it branches back, with `bx lr` and the caller continues (with subtracting `3` in this case).

Another way to look at this is that we pass `lr` as a special parameter to each function, so it knows where to return back.

**Well, actually…**

> I cannot help myself but mention the words *continuation-passing style* here. We won't cover it here, but it is a powerful compiler technique that makes passing the return address (or *continuation*) explicit early in the compiler pipeline.

## 7.18 Link register

The link register `lr` or `r14` is only special in the sense that `bl` works with `lr` and only `lr`. We could decide to use a different register for the return address, say `r8` (with the `mov r8, pc; b myLabel` sequence that we mentioned before), but `bl` is so convenient that we'll be using only `lr` for that purpose.

I must admit that the name *link register* is not perfect. It's called that because it creates a *link* that it can follow back to the callee. On some other architectures (like RISC-V), this register is called the *return address*, or `ra`.

## 7.19 Conditional execution and the CPSR register

So far, all the instructions we've covered executed *unconditionally*. As long as `pc` could reach those instructions, they were executed. However, sometimes we want *conditional execution*. This is when we want to decide whether to execute an instruction or not depending on some *condition*. This is done in two steps in ARM:

- A comparison instruction `cmp` compares two registers (or a register and an immediate), and saves the result of the comparison into the CPSR register.
- One or more instructions with a *condition code* reads the result of the comparison from CPSR and execute (or not), depending on the condition code.

CPSR stands for the *current program status register*.

Here's an example, where a bunch of `mov` instructions are executed depending on a condition:

```
cmp   r1, r2     /* cpsr = compare(r1, r2);    */
moveq r1, #10    /* if (eq(cpsr)) { r1 = 10; } */
```

```
mov   r2, #20    /* r2 = 20;                    */
moval r3, #20    /* r3 = 20;                    */
movne r4, #30    /* if (ne(cpsr)) { r4 = 30; } */
```

First, `cmp` instruction compares registers `r1` and `r2` and saves the result of the comparison into CPSR. It set a few bits in CPSR, but it's not important for us which exact bits are set, so in the pseudo-code, we've hidden that inside an opaque `compare` function.

Next, we see a bunch of instructions starting with `mov`. Those are: `moveq`, `moval`, `movne`. They are not separate instructions from `mov`, but are just `mov` with a condition code suffix. For example, `mov` with `eq` code executes if the last comparison made with `cmp` was equal; `mov` with `ne`—not equal, and so on. The `al` code stands for *always*, and is the default so it can be skipped, and thus, `mov` and `moval` is *exactly* the same instruction.

In the following table, you can see a summary of some condition codes. These are not all, but it's more than enough for our use case.

| Code | Operator | Description | Signed/Unsigned |
|---|---|---|---|
| `eq` | `==` | Equal | Either |
| `ne` | `!=` | Not equal | Either |
| `gt` | `>` | Greater than | Signed |
| `ge` | `>=` | Greater than or equal | Signed |
| `lt` | `<` | Less than | Signed |
| `le` | `<=` | Less than or equal | Signed |
| `hi` | `>` | Greater than | Unsigned |
| `hs` | `>=` | Greater than or equal | Unsigned |
| `lo` | `<` | Less than | Unsigned |
| `ls` | `<=` | Less than or equal | Unsigned |
| `al` | — | Always (default) | — |

The beautiful thing about the ARM instruction set is that these condition codes can be added to almost any instruction! Branch if equal? `beq`! Add if greater than? `addgt`! Compare—again!—if not equal? `cmpne`!

That's one of the features that makes the ARM instruction set *or-*

*thogonal*: conditional execution is available regardless of the instruction type.

## 7.20 Conditional branching

Combining branch instructions like `b` and condition codes gives us *conditional branching*. Conditional branching is used to implement `if`/`else` statements and loops like `while` and `for`.

## 7.21 Loader

How did our program get into memory? An operating system program called *loader* copied our program from disk into memory, and then set the program counter so that our program started execution from `main`.

> **Well, actually…**
>
> The real entry point is called `_start`, by convention, and it doesn't need to be a `.global`. But we linked our program with `libc`, and it defines its own `_start` entry point, which sets up things like program's command-line arguments, other machinery that is necessary for other `libc` functions to work. In turn, it will call `main`, and that is why it needs to be declared `.global`.

If the loader is an operating system program, then how did the operating system get into memory? That is done with the help of a program called *boot loader*.

> **Well, actually…**
>
> In embedded systems, the program is often stored in read-only memory that doesn't get erased on power-off, and when the system starts, the program counter starts executing from address 0.

## 7.22 Data and code sections

Let's look again at the data and code sections of our hello-world program.

```
.data
  hello:
    .string "Hello, assembly!"

.text
  .global main
  main:
    push {ip, lr}
    …
```

The data section declared with `.data` assembly directive is a span of memory that you are allowed to:

- read,
- write,
- but not execute.

The code section declared (confusingly) with `.text` is another span of memory that you are allowed to:

- read,
- but not write, and
- execute.

Why do we need separate sections? Why the restrictions on writing and executing?

The answer is many-fold.

First, it is a security feature of many operating systems: forbid data section from executing and forbid changing the code section. This makes it more complicated to inject malicious code by tricking the program to change it's code, or to jump to an area that was not supposed to be executed. It also helps with processor caches and running multiple instances of one program sharing the same code but using different data.

Even simpler, in embedded systems, the code section will most often go into read-only memory, while the data section will be the main read-write memory.

## 7.23 Segmentation fault

In our hello-world program, we don't modify the greeting string, do we? Does that mean that we can put it under the `.text` section?

Yes, we can even omit the `.text` section, since it's implicit:

```
hello:
  .string "Hello, assembly!"

.global main
main:
  push {ip, lr}

  ldr r0, =hello
  bl printf

  mov r0, #41
  add r0, r0, #1

  pop {ip, lr}
  bx lr
```

Try it out!

Oh, no! Our program crashes with a *segmentation fault*! What happened? What's segmentation fault anyway?

It can happen when we try to write to a read-only *segment*. But, in general, this term is applied to all kinds of *memory access violations*.

So what did we violate here? Alignment! Our string—`"Hello, assembly!"`—is 17 bytes long (including the implicit zero-terminator `\0`). So it pushed our `main` function to a memory address that is not an aligned word. We can pad the string with zeros, so it occupies 20 bytes (divisible by four):

```
hello:
  .string "Hello, assembly!\0\0\0"
```

Or, we can use an assembly directive called `.balign` to align it at a four-byte (word) boundary:

```
hello:
  .string "Hello, assembly!"
  .balign 4
```

> There's also an `.align` directive, but its exact meaning is inconsistent across architectures and assemblers. On the other hand, `.balign n` always aligns at a boundary specified as a number of *bytes*.

## 7.24 Data directives

One way to look at an assembly language, it is a way to encode binary data. As we know, each ARM instruction is encoded into a single machine word. However, the assembler has directives that allow us to encode literal data as well.

Two directives that we'll talk about are `.string` and `.word`, but there are more available.

We've already seen the string directive. One notable feature of it is that it encodes a zero-terminated string. That is, the string is padded with a single zero byte: `\0`.

A `.word` directive allows us to insert a literal machine word into our binary using a numeric notation:

```
fourtyTwo:
  .word 42
```

It's a good idea to put a label before a data directive so that we can refer to it later, but it is not always necessary, as you'll see below.

What are data directives good for? They can be used as constants or global variables in our program, but they are simply another encoding for binary data. Consider the following program.

```
.global main
main:
  .word 0xE3A0002A /* Same as `mov r0, #42` */
  bx lr
```

This is a short program that simply exits with code 42. But here, instead of writing `mov r0, #42`, we wrote `.word 0xE3A0002A`, which encodes the same value, as the instruction. As you might remember from instruction encoding, the last byte of an instruction encodes an immediate, which is `0x2A` in this case (`42` in decimal).

Writing instructions this way *is definitely a bad idea*, but it is a way for us to dispel the magic of what the assembler does for us.

## 7.25 Loading data

Loading means copying data from memory into one or more registers. We've already seen how to load data, given a label. For

example, load a string address and pass it to `printf` to be printed to the console.

```
ldr r0, =hello
bl printf
```

Similarly, you can load an address of a word. But this way, you only get the address of the word, and not the word itself.

Once we have loaded an address of a word, we can load the word itself. For that we use another form of `ldr` instruction, with square brackets:

```
ldr r1, =myWord    /* r1 = myWord */
ldr r0, [r1]       /* r0 = M[r1]  */
```

Assuming we have a word with that label stored somewhere:

```
myWord:
  .word 42
```

When we load an address into a register, we call it a *pointer*. This pointer *references* another word, but it's not that word itself. When we load the word that is referenced by that pointer, we say that we *dereference* a pointer.

In this example, when the two instructions execute, `r1` will contain some seemingly random number, which is the address corresponding to the label `myWord`, while `r0` will contain the loaded value itself, that is `42`.

In the listing, we used a new notation for our pseudo-code. We used `M[r1]` to refer the memory word located at the address found in `r1`. Remember, how we said that memory is like a large array? This notation is easy to remember because the square brackets in the instruction syntax (`[r1]`) match the square brackets of our notation (`M[r1]`). This will always be the case.

However, loading a single word from a location known ahead of time is not very interesting. For that there's a shortcut:

```
ldr r1, =42    /* r1 = 42 */
```

Even better, if the constant that we are loading can fit into an immediate value, the assembler will translate this into:

```
mov r1, #42    /* r1 = 42 */
```

However, if it doesn't fit into an immediate, it will convert it into something roughly equivalent to our previous example:

```
    ldr r1, =temporaryWord    /* r1 = temporaryWord; */
    ldr r1, [r1]              /* r1 = M[r1]; */
    …

  temporaryWord:
    .word 42
```

This version of `ldr` instruction is called a *pseudo-instruction*, because, as we have just seen, it will be converted by the assembler to one or more different instructions.

## 7.26 Load with immediate offset

It is not necessary to have a label for each word to load it. You can also load a word given an *offset* from another word.

For example, if we have several consequtive words:

```
  myArray:
    .word 42
    .word 44
    .word 46
```

(Which we can also write as follows:)

```
  myArray:
    .word 42, 44, 46
```

Then, we can load the third word using the following syntax:

```
    ldr r1, =myArray       /* r1 = myArray;   */
    ldr r0, [r1, #8]       /* r0 = M[r1 + 8]; */
```

Here we loaded a word given an 8-byte (and thus, 2-word) offset. As a result, we have loaded `46` into `r0`.

The offset can be positive or negative. It can be an immediate (like in this case) or a register. For example:

```
    ldr r0, [r1, -r2]    /* r0 = M[r1 - r2]; */
```

## 7.27 Storing data

Storing means copying data from one or more registers into memory. In other words, we are modifying, or *mutating* memory.

As we already mentioned, it is not possible to store data into the read-only segment that we declare with the `.text` directive: this will lead to a segmentation fault.

However, we can store data into the `.data` segment, as well as into stack and heap (that we'll get into in a short while).

Store instructions use `str` mnemonic, and use the same syntax as `ldr` instruction. Some examples (with `ldr` for comparison):

```
str r0, [r1]          /* M[r1] = r0; */
ldr r0, [r1]          /* r0 = M[r1]; */

str r0, [r1, #8]      /* M[r1 + 8] = r0; */
ldr r0, [r1, #8]      /* r0 = M[r1 + 8]; */

str r0, [r1, -r2]     /* M[r1 - r2] = r0; */
ldr r0, [r1, -r2]     /* r0 = M[r1 - r2]; */
```

The square brackets always remind us which part is for the address.

## 7.28 Stack

The *stack*, or a *call stack*, is used to implement nested, potentially recursive function calls. It is a segment of memory, not unlike the code segment and the data segment.

In computing, a *stack* usually refers to a particular data structure. The word *stack* is used as a metaphor for stacks of things, like books. If you have a stack of books, it is easy to add one or a few books at the top, and it is easy to remove one or a few books at the top. However, it is trickier to add or remove books in the middle or at the bottom of the stack. A *stack* is a data structure that makes it easy (and efficient) to add and remove items from one end.

In JavaScript you rarely see anyone defining such data structure. This is because the built-in `Array` is good enough as a stack: it has a `push` method to add the last element, and a `pop` method to remove it, and they are reasonably efficient.

The *call stack* is not a data structure in the sense of `Array`. It is a segment of memory. However, unlike those segments, it requires us to follow a particular discipline or convention to use it. This convention is called the *calling convention*. It is defined so that functions written in different languages, or using different compilers can call each other and interoperate on some basic level.

For example, when we call some `libc` functions like `printf`, we need to follow this convention. If we want other functions to call a function we defined (as in the case of `main`), we also need to follow it.

First, we will look at the basic stack operations: push and pop, and then we will discuss the calling conventions, and how it can be implemented using these operations.

## 7.29 Push and pop

The *stack pointer* is the central concept for the call stack. In ARM assembly, it is referred to as `r13` or, more often as `sp`. It holds the address of the *top of the stack*. As the stack pointer changes its value, we say that the stack grows or shrinks. The stack grows towards smaller addresses of the memory, and shrinks towards larger addresses. We say that a word is allocated on the stack if it is pointed by the stack pointer, or below.

The following instruction pushes four words onto the stack.

```
push {r0, r1, r2, r3}
```

In the next figure, you can see several registers and a stretch of memory belonging to the call stack, shown in two states. On the left, you can see the state before executing the push instruction, and on the right, you can see the state after executing it.

The effect is that the words are copied from registers to the stack, and the stack pointer is *decremented* by 16 bytes, and thus, four words. It is *decremented* because it grows towards lower addresses. The area above the stack pointer is greyed out to signify that this area is not allocated.

We could have acheived the same effect using the following instructions:

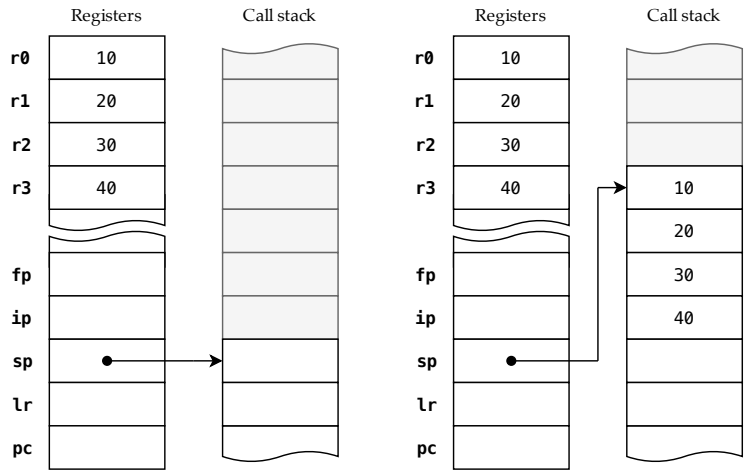


Figure 7.8: Call stack before and after executing: `push {r0, r1, r2, r3}`

```
sub sp, sp, #16
str r0, [sp, #0]
str r1, [sp, #4]
str r2, [sp, #8]
str r3, [sp, #12]
```

Or the following sequence of individual `push` instructions:

```
push {r3}
push {r2}
push {r1}
push {r0}
```

The `push` instruction has a counterpart, `pop`:

```
pop {r0, r1, r2, r3}
```

This reverses the previous action: *increments* `sp` by 16, and copies values from stack into registers. It does this in the reverse order, in other words, like this:

```
pop {r0}
pop {r1}
pop {r2}
pop {r3}
```

These instructions use curly braces that remind us of the set notation. This is to signify that the order in which you write them in the assembly does not matter.

You can write it either way:

```
push {r3, r1, r2, r0}  // Same as: push {r0, r1, r2, r3}
```

But the effect will be the same:

- Lower registers (think `r0`) are pushed to and popped from the lower memory addresses.
- Higher registers (think `r15`) are pushed to and popped from the higher memory addresses.

The `push` and `pop` instructions' encodings have 16 bit-flags dedicated to the 16 registers. So they can express which registers to push or pop, but not their order. The order was picked so that a pop operation would undo the effect of the corresponding push operation.

## 7.30 Stack alignment

An essential peculiarity of the call stack is that the stack pointer should be aligned at the 8-byte (2-word) boundary at so-called *external interfaces*. In this book, we won't discuss what this means and will just mention that calling a `libc` function is one of those situations. Our compiler will not distinguish between calling a `libc` foreign function from a baseline language function. For simplicity, we will assume an even stricter requirement that *the stack pointer should always be 8-byte (2-word) aligned*.

So, what if we need to push a single word onto the stack? If we just do `push {r4}`, then our stack won't be aligned. Thus, it is common to push some other register as a dummy to maintain the stack alignment when needed:

```
push {r4, ip}
```

Here we used the `ip` registers for that purpose. The same as we did when we defined our `main` function:

```
.global main
main:
  push {ip, lr}
  …
```

In general, we can always pad a push with a dummy register to maintain the stack alignment. When constructing the compiler, we will also talk about how we can maintain the stack alignment without wasting space unnecessarily.

## 7.31 Arguments and return value

Function arguments are passed in the first four registers: `r0`–`r3`. The rules are elaborate regarding differently-sized arguments, but in our compiler, all values are represented using a single word. Thus, we can say that the first four arguments are passed in the registers `r0`–`r3`. These registers are also sometimes called the *argument registers*.

The rest of the arguments are passed by pushing them onto the stack.

For example, if a function `f` takes six parameters, to make a call `f(10, 20, 30, 40, 50, 60)` we need to write the following assembly:

```
mov r0, #50
mov r1, #60
push {r0, r1}   // 50 and 60 go on stack
mov r0, #10     // 10, 20, 30, 40 go in registers
mov r1, #20
mov r2, #30
mov r3, #40
bl f            // The actual call
add sp, sp, #8  // Deallocate 2 words of stack
```

It is also the caller's responsibility to deallocate the stack space used for the arguments that didn't fit into the four argument registers. After a function returns, the return value (if any) is expected to be in `r0`. (It can also span `r0`–`r1` in case of a 64-bit return value.)

It is expected that the assembly programmer or the compiler writer knows (from external sources) about the correct signature of each function: the number of parameters, their size, and about the return value (if any). For a `libc` function, we can look them up in the documentation, and for a function defined in our language, we can see it from its definition.

## 7.32 Register conventions

After a call returns, the contents of registers `r0`–`r3` will be unpredictable and semi-random and should not be relied upon. (This is, of course, except for the registers that are used for the return value, if any.) In other words, these registers will contain *garbage* that is left over after the function call: the called function itself might have used them or called another function. We say that the registers `r0`–`r3` are *call-clobbered*. Another register that is a *call-clobbered* register is `r14` or `lr`. It is clobbered by the design of the "call" instruction `bl`, which overwrites the `lr` with the return address. Another *call-clobbered* register is `r12` or `ip`, which is clobbered by the linker-generated veneer.

In contrast, all other registers are *call-preserved*. In other words, each function's responsibility is that the values in these registers are preserved when a call returns. Can a function modify them at all? Yes, but it should make sure to restore the values before returning from the call.

If *call-clobbered* registers are a good fit for arguments, the *call-preserved* registers are a good fit for variables. The registers `r4` to `r10` are sometimes referred to as *variable registers*. Other *call-preserved* registers are: frame pointer and stack pointer (`r11` or `fp` and `r13` or `sp`). (We'll get to the frame pointer in a minute.) It is expected that the top of the stack will be at the same address as before the call. The program counter register (`r15` or `pc`) is a special register, but you can think of it as being call-preserved too: when the call returns back, the `pc` is restored to the previous value.

So, how do we preserve the values of the call-preserved registers, in practice? When the call starts, we decide which call-preserved registers we want to use, so we push them onto the stack. Then we proceed with using them to our liking. Then we pop them off the stack before returning the call. It is also said that we are *saving* these registers on the stack.

Thus, the terms *call-clobbered* and *call-preserved* have an alternative terminology: *callee-saved* and *caller-saved*. This refers to which side of the call is responsible for saving the registers onto the stack, *if they want them to be preserved*. So that's another way to look at it.

Register conventions are summed up in the following table.

| Register | Role | Convention |
|---|---|---|
| `r0` | Argument/return register | *Call-clobbered* |
| `r1` | Argument register | *Call-clobbered* |
| `r2` | Argument register | *Call-clobbered* |
| `r3` | Argument register | *Call-clobbered* |
| `r4` | Variable register | *Call-preserved* |
| `r5` | Variable register | *Call-preserved* |
| `r6` | Variable register | *Call-preserved* |
| `r7` | Variable register | *Call-preserved* |
| `r8` | Variable register | *Call-preserved* |
| `r9` | Variable register | *Call-preserved* |
| `r10` | Variable register | *Call-preserved* |
| `r11` or `fp` | Frame pointer | *Call-preserved* |
| `r12` or `ip` | Intra-procedure scratch register | *Call-clobbered* |
| `r13` or `sp` | Stack pointer | *Call-preserved* |
| `r14` or `lr` | Link register | *Call-clobbered* |
| `r15` or `pc` | Program counter | *Call-preserved* |

## 7.33 Frame pointer

The last register that we haven't talked about is the *frame pointer* denoted as `r11` or `fp` in assembly. A function's *frame* is an area of the stack which is allocated by that function. It is also referred to as *stack frame*, *call frame*, or *activation record*. When it is used, the *frame pointer* is set to point to the *base* of the frame, in other words, to an address where the stack pointer used to point when the function was called. The area of the stack between the stack pointer and the frame pointer is the current function's stack frame. The frame pointer is a call-preserved register. By convention, we save the frame pointer in the same stack slot where the new frame pointer points. This way, the frame pointer register gives a start to a *linked list* of stack frames, where intermediate links are the stack-saved frame pointers of the callers. See the diagram that follows.

Unlike in the figure, each stack frame usually has a different size, depending on a number of local variables and intermediate values.

It is optional to use the frame pointer register this way, and it is optional to maintain the linked list of stack frames. However, it is used by debuggers, introspection tools, and it may be necessary for certain language features, such as exception handling and non-

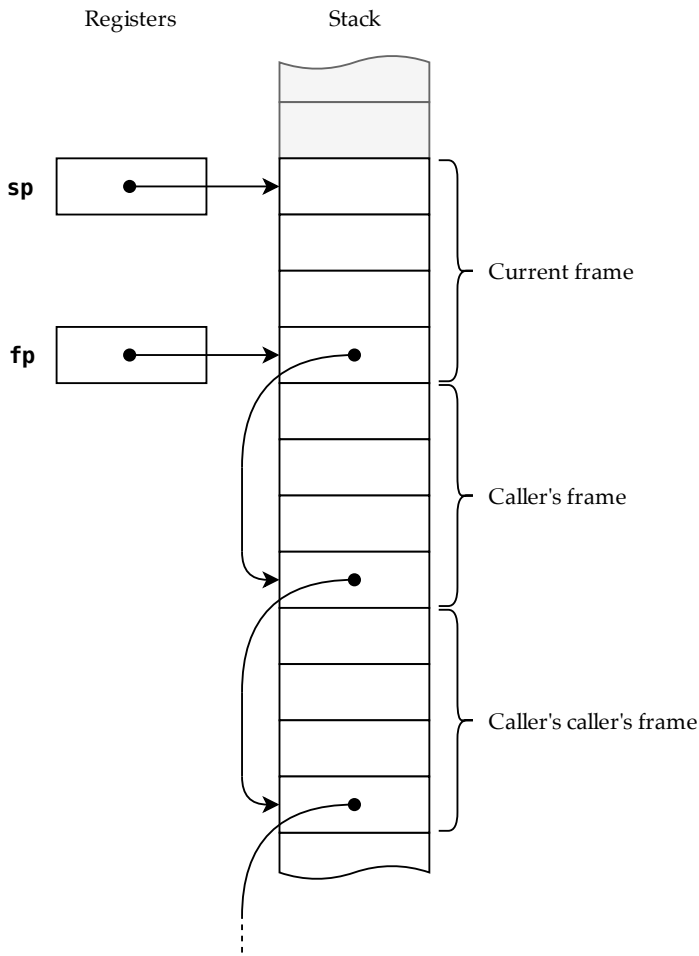


Figure 7.9: Example of a stack with several frames

local go-to statements. Since the frame pointer doesn't change after the function is called, it is a convenient pointer to use when loading data from the stack frame, such as local variables. You know that `fp + 4` always refers to the same stack slot, where a variable can be stored. At the same time, loading local variables relative to the stack pointer is a bit tricky: stack pointer often changes its value, but it is certainly possible. The baseline compiler we will use the frame pointer, but in *Part II*, we will talk about how we can get away without a frame pointer, and thus free up another variable registers.

## 7.34 Function definitions

We have already defined some functions previously, `main` and `addFourtyTwo`. The latter looks pretty simple:

```
addFourtyTwo:
  add r0, r0, #42
  bx lr
```

Calling this function doesn't allocate any stack at all! Neither for intermediate values nor for saving registers. The return address is passed in the `lr` registers. And it doesn't need to save any registers, because it doesn't use any call-preserved registers.

This all changes as soon as you want to define a function that calls another function, like the `main` function, that we have seen before:

```
.global main
main:
  push {ip, lr}
  ldr r0, =hello
  bl printf
  mov r0, #41
  add r0, r0, #1
  pop {ip, lr}
  bx lr
```

Since it calls `printf`, this call will potentially clobber all the call-clobbered registers, notably: the return address `lr`. If we want to be able to return from `main`, we better save this register onto the stack. And that's exactly what we did in our `main` definition by doing `push {ip, lr}`. We have also saved a dummy `ip` register to maintain stack-alignment.

Now, if we wanted to maintain the frame pointer and the linked-list of frames, we would do things differently. We would push both `lr` and `fp` onto the stack and then set the new frame pointer to match the stack pointer. Now we can push and pop the stack, and we will still have a solid frame of reference in `fp` for accessing local variables. Using the `fp` also makes it easy to deallocate the frame. You can do it with `mov sp, fp` at the end of your function. Alternatively, you could try to match all the pushes and pops, but it can be tricky. All this busywork has a name, or two in fact: function's *prologue* and *epilogue*.

```
.global main
main:
  push {fp, lr}  // Function's
  mov fp, sp     // prologue
  …
  mov sp, fp     // Function's
  pop {ip, lr}   // epilogue
  bx lr          //
```

Function's prologue saves the call-preserved registers, sets up the frame pointer, if necessary. Function's epilogue reverses this and exits the function.

In this epilogue, we `pop` the registers back, and return using `bx lr`. If you remember, in this case, `bx lr` is the same as `mov pc, lr`. So, why not pop the return address directly into `pc`, instead of `lr`? This way, we can make our epilogue shorter and more efficient:

```
  mov sp, fp     // Function's
  pop {ip, pc}   // epilogue
```

This is a common way to organize a function's prologue and epilogue in ARM.

## 7.35 Heap

The other memory segment is called the *heap* or the *dynamic memory*. It is dynamic in that a program can ask the operating system to allocate or deallocate a new region of memory at runtime. This is usually done by calling the `libc` functions (`malloc` and `free`), which eventually make the system calls for you. The function `malloc` takes a single argument into `r0` that specifies how many bytes we want to allocate. It returns a pointer into `r0`, which points to the

newly allocated region of memory. The function `free` takes such pointer as its single parameter in `r0` and deallocates that region of memory. After that, you can't use this region, and the operating system will use that memory somewhere else.

Here's a small snippet of assembly which allocates a single word on the heap, stores the number `42` into it, and then deallocates it. It uses a similar-style prologue and epilogue to what we have seen before.

```
.global main
main:
  push {ip, lr}    // Prologue

  mov r0, #4       // Allocate four
  bl malloc        // bytes (one word)

  mov r3, #42      // Store 42 into
  str r3, [r0, #0] // that word

  bl free          // Free/deallocate

  pop {ip, pc}     // Epilogue
```

It is common in high-level languages that most of the objects and data structures are allocated on the heap, and the functions are only passing and returning pointers to these objects.

---

There's much more to discuss about ARM assembly and assembly programming, in general. But this chapter should be enough to get a solid grasp on it. And even more than enough to complete the second pass of our compiler.

# Chapter 8

# Second Pass: Code Generation

The second and last pass of our compiler is called *emitter* or *code generator*, since it generates the assembly code. It converts an AST of a program into assembly instrucitons. How do we organize that? We extend the `AST` interface with a new method called `emit`.

```
interface AST {
  emit(): void;
  equals(AST): boolean;
}
```

It's a cool method—you say—it takes no parameters and returns nothing (or `void`). So, what's the use of it? First, it takes the implicit `this` parameter, and with it all the instance variables of each node. Second, even though the signature return type is `void`, when this method is called, it will emit assembly as a *side effect*. For that we will use an `emit` function (as opposed to a method). Using this function we can emit instructions like this:

```
emit("  add r1, r2, r3");
```

More often, we will use template literals for string interpolation when emitting code:

```
let x = 42;
emit(`  add r1, r2, #${x}`);
```

This will emit `add r1, r2, #42`. How do we implement the `emit` function? We can define it so that it appends a line to an array, or writes a line to a file. But for now, let's define it as follows, for simplicity:

```
let emit = console.log;
```

That's right, `emit` simply prints to the console's standard output channel. This is good enough, for now. We can redirect the standard output channel to a file, and then assemble it.

Now, what about the *method* `emit`? First, we define this method on every AST node to satisfy the interface. A dummy implementation will suffice, for a start. Here we use `Number` node as the example:

```
class Number implements AST {
  constructor(public value: number) {}

  emit() {
    throw Error("Not implemented yet");
  }

  equals(other: AST) {…}
}
```

Next, we will modify each node to emit the correct assembly. For example, in case of `Number` it will eventually be as follows:

```
  emit() {
    emit(`  ldr r0, =${this.value}`);
  }
```

This way, each AST node will be able to emit corresponding assembly. Nodes that have other nodes as instance variables will be able to call their `emit` methods. Note that the order of calls matters.

There's one rule that we will follow: each node, when emitted, will produce assembly code that will put its result into register `r0`. This is alredy how function calls work, but we extend it for all nodes that produce a value. This way, we will know where each value ends up when we emit it.

## 8.1 Test bench

To test our emitter pass as we develop it, I suggest adding a few temporary AST nodes: `Main` and `Assert`. As we teach our compiler to emit each kind of node, we want to see the results immediately, not until we can define a somewhat complicated function like `assert`, so we make it into a primitive. It's the same with `Main`: we want our compiler to produce an entry point to our program before we can define functions. We define `Main` to take an array of statements. It will arrange an entry point and an exit point for them.

```
class Main implements AST {
  constructor(public statements: Array<AST>) {}

  emit() { /* TODO */ }

  equals(other: AST) {…}
}
```

While `Assert` takes a single `condition` on which it will make the assertion:

```
class Assert implements AST {
  constructor(public condition: AST) {}

  emit() { /* TODO */ }

  equals(other: AST) {…}
}
```

We've left out the `emit` methods for now.

We sligtly modify the `functionStatement` parser to temporarily produce the `Main` node, in case the function name is `main`:

```
// functionStatement <-
//   FUNCTION ID LEFT_PAREN parameters RIGHT_PAREN
//  blockStatement
let functionStatement: Parser<AST> =
  FUNCTION.and(ID).bind((name) =>
    LEFT_PAREN.and(parameters).bind((parameters) =>
      RIGHT_PAREN.and(blockStatement).bind((block) =>
        constant(
          name === 'main'
```

```
            ? new Main(block.statements)
            : new Function(name, parameters, block)))));
```

Same with `Assert`: we produce it inside the `call` parser in case the callee is called `assert`.

```
// call <- ID LEFT_PAREN args RIGHT_PAREN
let call: Parser<AST> =
  ID.bind((callee) =>
    LEFT_PAREN.and(args.bind((args) =>
      RIGHT_PAREN.and(constant(
        callee === 'assert'
          ? new Assert(args[0])
          : new Call(callee, args))))));
```

As soon as we have enough functionality to define our own functions like `assert`, we'll be able to roll this back.

## 8.2 Main: entry point

Let's imagine how our main entry point should look like in assembly.

```
.global main
main:
  push {fp, lr}
  /* … */
  mov r0, #0
  pop {fp, pc}
```

As before, we need a `main` label and to declare it `.global`. We won't strictly need it until we implement calls, but let's save the `lr` register, so it won't get clobbered when we implement calls, and we can safely return from `main`. Since we need to align the stack at the double-word boundary, we can push some other register together with `lr`. We could push `ip` as a dummy, but why not as well push the frame pointer `fp` to get a "classical" function prologue going. Then we would generate the inner statements. We set the return value to zero, which will be the default return code of the program. We finish with a "classical" epilogue, where we restore `fp` and by pushing the saved `lr` into `pc` we return from `main` as well.

Now we can implement the `emit` method of `Main`:

```
class Main implements AST {
  constructor(public statements: Array<AST>) {}

  emit() {
    emit(`.global main`);
    emit(`main:`);
    emit(`  push {fp, lr}`);
    this.statements.forEach((statement) =>
      statement.emit()
    );
    emit(`  mov r0, #0`);
    emit(`  pop {fp, pc}`);
  }

  equals(other: AST) {…}
}
```

Inbetween the prologue and epilogue, we emit the inner statements in order using `forEach` loop. Though, we haven't defined any of the statement nodes that could be emitted here yet. Nevertheless, our `Main` node is enough to make the first test for our emitter pass: a program that does nothing successfully!

When testing the emitter pass, we could do that in isolation:

```
new Main([]).emit();
```

Or we could integrate it with the parser:

```
parser.parseStringToCompletion(`
  function main() {

  }
`).emit();
```

We can pipe the resulting assembly into a file, assemble it and execute it!

Congratulations, our compiler has just compiled its first program end-to-end!

## 8.3 Assert

Like a function call, assert will expect its only argument to be located in `r0` register. We compare it with `1` which we treat as a

truthy value in our untyped language with no proper booleans. If equal, we save an ASCII code for dot into `r0` to signify success. Otherwise, we save the code for `F` to signify failure. We call `putchar` from `libc` to print it. We do not terminate the program.

```
cmp r0, #1
moveq r0, #'.'
movne r0, #'F'
bl putchar
```

To implement the `Assert` node, we just copy-paste this assembly into the `emit` method. We also make sure to emit the condition, so the result ends up in `r0`. We don't have any nodes that could be emitted here yet, but that will change quickly.

```
class Assert implements AST {
  constructor(public condition: AST) {}

  emit() {
    this.condition.emit();
    emit(`  cmp r0, #1`);
    emit(`  moveq r0, #'.'`);
    emit(`  movne r0, #'F'`);
    emit(`  bl putchar`);
  }

  equals(other: AST) {…}
}
```

## 8.4 Number

Let's start with the node for a literal integer. This way, we can put our `Assert` to use as soon as possible.

We need to load the integer value into `r0`. We could use `mov` with an immediate, but the range of immediate values is quite restrictive. That's why we use the `ldr` pseudo instruction. As you remember, the assembler will convert it into `mov` with an immediate, if possible.

```
class Number implements AST {
  constructor(public value: number) {}

  emit() {
```

```
    emit(`  ldr r0, =${this.value}`);
  }

  equals(other: AST) {…}
}
```

Now, we can use `Number` in an assertion.

```
parser.parseStringToCompletion(`
  function main() {
    assert(1);
  }
`).emit();
```

We compile, assemble, and run this program, and we can see that it prints a mighty dot, signifying that the assertion passes. If you change the integer to 0, you can see the program prints `F`, as it should.

## 8.5 Negation

Next up is the negation operator! It is, to some degree, similar to the `Assert`. It takes a single term, that we emit, and compares it to zero. Depending on that, we move either `1` or `0` into the `r0`, thus logically negating the result.

```
class Not implements AST {
  constructor(public term: AST) {}

  emit() {
    this.term.emit();
    emit(`  cmp r0, #0`);
    emit(`  moveq r0, #1`);
    emit(`  movne r0, #0`);
  }

  equals(other: AST) {…}
}
```

Now we can extend our test program:

```
function main() {
  assert(1);
```

```
    assert(!0);
  }
```

From here on, we will skip the boilerplate when we discuss our test program.

## 8.6 Infix operators

Next up are infix operators: `==`, `!=`, `+`, `-`, `*`, `/` with nodes `Equal`, `NotEqual`, `Add`, `Subtract`, `Multiply`, `Divide`. All of them are very similar: they take two terms `left` and `right` and operate on them.

Let's use `Add` as an example.

We could emit one node, move the value into a temporary register (`r1`, in this case), then emit the second node (which value end up in `r0`) and then sum them up:

```
class Add implements AST {
  constructor(public left: AST, public right: AST) {}

  emit() { /* First flawed attempt */
    this.left.emit();
    emit(`mov r1, r0`);
    this.right.emit();
    emit(`add r0, r0, r1`);
  }

  equals(other: AST) {…}
}
```

However, this will not work for long: as soon as the `right` node is something more complicated than just an `Number`, emitting it will likely clobber `r1` that stores the `left` result. This will be the case, for example, if we have two nested infix nodes. That's why we need to save `r1` to the stack before emitting `right`, then restore it before we perform the addition:

```
emit() { /* Second attempt */
  this.left.emit();
  emit(`push {r0, ip}`);
  this.right.emit();
  emit(`pop {r1, ip}`);
```

```
    emit(`add r0, r0, r1`);
  }
```

To maintain 8-byte stack alignment, we also save/restore the dummy `ip` register. If the `left` node is another nested expression, the pushes and pops will match, and we will be at the same stack pointer location before and after we emit it.

Like in our first attempt, we can avoid using stack in some cases. We can also do better than waste stack space on saving `ip` each time. We will explore it further in *Optimizations* chapter in *Part II*.

All our infix operators will have the same structure: emit right, push, emit left, pop, then the action. Only the last part will be different. So, here is a quick table that shows the action part of each infix node.

| AST node | Emits |
|---|---|
| `Add` | `add r0, r0, r1` |
| `Subtract` | `sub r0, r0, r1` |
| `Multiply` | `mul r0, r0, r1` |
| `Divide` | `udiv r0, r0, r1` |
| `Equal` | `cmp r0, r1`<br>`moveq r0, #1`<br>`movne r0, #0` |
| `NotEqual` | `cmp r0, r1`<br>`moveq r0, #0`<br>`movne r0, #1` |

If you want to reduce duplication, you can pull the common part into a new AST node called `Infix`, for example.

In the end, we can add an integration test for infix operators:

```
function main() {
  assert(1);
  assert(!0);
  assert(42 == 4 + 2 * (12 - 2) + 3 * (5 + 1));
```

```
}
```

We are getting somewhere!

## 8.7 Block statement

Block statement is similar to `Main` but without all the entry point fuss. It merely emits each statement.

```
class Block implements AST {
  constructor(public statements: Array<AST>) {}

  emit() {
    this.statements.forEach((statement) =>
      statement.emit()
    );
  }

  equals(other: AST) {…}
}
```

Simple test:

```
function main() {
  assert(1);
  assert(!0);
  assert(42 == 4 + 2 * (12 - 2) + 3 * (5 + 1));
  { /* Testing a block statement */
    assert(1);
    assert(1);
  }
}
```

## 8.8 Function calls

Next is function calls. As we learned in the previous chapter, according to the ARM calling convention we need to put the first four arguments into registers `r0`–`r3`, and expect the return value in `r0`. Let's remind ourselves that the `Call` node holds a `callee` string and an `args` array.

```
class Call implements AST {
  constructor(public callee: string,
```

```
                public args: Array<AST>) {}

  emit() {
    …
  }
  …
}
```

Let's start with something simple: calling a function with no arguments. That's just branching-and-link to the callee label:

```
// One argument
emit(`  bl ${this.callee}`);
```

How about one argument? If we emit the first argument, then it will be in `r0`. And that's precisely where we need it! Then we `bl` to it, as previously:

```
// Two arguments
this.args[0].emit();
emit(`  bl ${this.callee}`);
```

More arguments? We could have special cases for two, three, and four arguments, and it would also be good for performance reasons, but how about we jump straight to the more general case. The following will handle two, three, or four arguments. It was nice not to use any stack space for one or two arguments. However, to evaluate more arguments, we need to put them on the stack temporarily. Otherwise, as we evaluate one, we risk clobbering the value of the other arguments. Here's how we do it:

```
// Up to four arguments
emit(`  sub sp, sp, #16`);
this.args.forEach((arg, i) => {
  arg.emit();
  emit(`  str r0, [sp, #${4 * i}]`);
});
emit(`  pop {r0, r1, r2, r3}`);
emit(`  bl ${this.callee}`);
```

First, we allocate enough stack space for up to four arguments, or in other words, 16 bytes. We do that by subtracting from the stack pointer since the stack grows from higher memory addresses to lower. Then we loop through arguments using `forEach`, which takes a callback with each item, and optionally its index `i`. For each

argument, we first emit it and then store the result into each stack slot. We multiply by four to convert array indexes `0`, `1`, `2`, `3` into stack offsets in bytes: `0`, `4`, `8`, `12`. At this point, the arguments will all be emitted and stored in the stack. However, we want them to be in the registers. So we `pop` them into the registers in one elegant go. Now, we've got everything in place so we can branch to the label.

Here's the complete version that combines all three approaches by dispatching on the number of arguments.

```
class Call implements AST {
  constructor(public callee: string,
              public args: Array<AST>) {}

  emit() {
    let count = this.args.length;
    if (count === 0) {
      emit(`  bl ${this.callee}`);
    } else if (count === 1) {
      this.args[0].emit();
      emit(`  bl ${this.callee}`);
    } else if (count >= 2 && count <= 4) {
      emit(`  sub sp, sp, #16`);
      this.args.forEach((arg, i) => {
        arg.emit();
        emit(`  str r0, [sp, #${4 * i}]`);
      });
      emit(`  pop {r0, r1, r2, r3}`);
      emit(`  bl ${this.callee}`);
    } else {
      throw Error(
        "More than 4 arguments are not supported");
    }
  }

  equals(other: AST) {…}
}
```

We throw an error in case of more than four arguments, which we don't support in the baseline compiler.

**Explore…**

> By specializing code generators for two, three, and four arguments separately (like we did for zero and one argument) you will be able to produce better code (for example, not allocate four slots for two arguments, like we do now). From the previous chapter you might have an idea about how to handle more than four arguments.

How do we test this? We don't yet have a way to define functions. But we can use some of the `libc` functions to test this.

As you remember, `putchar` takes one parameter: let's put it into use by printing the ASCII code for a dot (46), as our test.

There's also a function called `rand` that takes no arguments and returns a random number. That's not the best function to base our tests on, but we can use it until we can define our own functions.

```
putchar(46);
assert(rand() != 42);
```

I couldn't find any portable `libc` function that takes two, three, or four integer arguments, so we'll have to wait to test that.

> **Explore…**
>
> Modify the parser such that a character literal like `'.'` produces an AST node with the corresponding ASCII code, like `new Number(46)`. You can use `string.charCodeAt(index)` method.

## 8.9 If-statement

Next one is fun: conditional statement, or, in other words, if-statement. It is fun because we are finally getting into working with control-flow.

Let's look at an example first. What if we wanted to compile the following if-statement:

```
if (rand()) {   /* Conditional */
  putchar(46);  /* Consequence */
} else {
  putchar(70);  /* Alternative */
}
```

How would we compile this by hand? We could start with two labels `ifTrue` and `ifFalse`. We branch to `ifFalse` if the condition is zero, otherwise to `ifTrue`. Below each of these labels, we emit the code corresponding to that branch. In the end, we put an `endIf` label and make sure that both branches jump to it when they reach their end. Here's what we get:

```
    bl rand
    cmp r0, #0
    beq ifFalse
    bne ifTrue

  ifTrue:
    ldr r0, =46
    bl putchar
    b endIf

  ifFalse:
    ldr r0, =70
    bl putchar
    b endIf

  endIf:
    /* Whatever follows next */
```

We can do precisely that, but there are few improvements that we can make, with respect to efficiency. First, the `ifFalse` branch ends with `b endIf` immediately followed by `endIf:` label. That means we don't have to jump to it since it points to the next instruction anyway. Similar with `bne ifTrue`, but with one nuance: although this branch is conditional (`ne` suffix), it is in fact not. Since the two conditions `beq` and `bne` are mutually exclusive, if the execution reached `bne ifTrue` we already know that it will execute. Even more, since `bne ifTrue` is the only user of this label, we can completely get rid of `ifTrue`. Here's how the resulting code will look like:

```
    bl rand
    cmp r0, #0
    beq ifFalse

    ldr r0, =46
    bl putchar
```

```
    b endIf

  ifFalse:
    ldr r0, =70
    bl putchar

  endIf:
    /* Whatever follows next */
```

It is shorter, has fewer branches, so it can be executed more efficiently. But before we jump straight to implement this in our emitter, we need to solve one problem. We can't have two different labels called `ifFalse` or `endIf`: labels must be unique. We need to generate unique labels.

By convention, labels starting with prefix `.L`, for example, `.L123` are used for code generation. Let's create a class that will help us generate such labels.

```
class Label {
  static counter = 0;
  value: number;

  constructor() {
    this.value = Label.counter++;
  }

  toString() {
    return `.L${this.value}`;
  }
}
```

This class has a global `counter` which is incremented each time the constructor is called. When a `Label` object is created, this counter value is stored into the object's `value` instance variable. We define a `toString` method that adds the `.L` prefix, and so that these objects work well with string interpolation.

Now we have all things in place to implement the emitter. We crate two label objects `ifFalseLabel` and `endIfLabel` and use them with string interpolation in the `emit` calls.

```
class If implements AST {
  constructor(public conditional: AST,
              public consequence: AST,
```

```
                public alternative: AST) {}

  emit() {
    let ifFalseLabel = new Label();
    let endIfLabel = new Label();
    this.conditional.emit();
    emit(`  cmp r0, #0`);
    emit(`  beq ${ifFalseLabel}`);
    this.consequence.emit();
    emit(`  b ${endIfLabel}`);
    emit(`${ifFalseLabel}:`);
    this.alternative.emit();
    emit(`${endIfLabel}:`);
  }

  equals(other: AST) {…}
}
```

Now all of our if-statements will have unique labels!

We can throw in a small unit test, for good measure:

```
if (1) {
  assert(1);
} else {
  assert(0);
}

if (0) {
  assert(0);
} else {
  assert(1);
}
```

## 8.10 Function definition and variable look-up

It was nice that we could develop and test function calls in isolation: without even being able to define functions. However, the next two concepts are highly intertwined: function definitions and variable look-up. We already had a special node called `Main` that could be thought as a function that takes no parameters, and al-

ways returns `0`. We will use it as a template for making our `Function` node's emitter:

```
class Function implements AST {
  constructor(public name: string,
              public parameters: Array<string>,
              public body: AST) {}

  /* First draft */
  emit() {
    if (this.parameters.length > 4)
      throw Error(
        "More than 4 params is not supported");

    emit(``);
    emit(`.global ${this.name}`);
    emit(`${this.name}:`);
    this.emitPrologue();
    this.body.emit();
    this.emitEpilogue();
  }

  emitPrologue() {
    emit(`  push {fp, lr}`);
    emit(`  mov fp, sp`);
    emit(`  push {r0, r1, r2, r3}`);
  {}

  emitEpilogue() {
    emit(`  mov sp, fp`);
    emit(`  mov r0, #0`);
    emit(`  pop {fp, pc}`);
  }

  equals() {…}
}
```

Similar to `Call` we will limit ourselves with four parameters this time. We make a guard enforce this and raise an error otherwise. We start by emitting an empty line: to separate assembly functions for readability. Similarly to `Main`, we emit the `.global` directive and the label. However, this time we don't hardcode it to `main`, but use the name of the function instead, using string interpolation.

Here the function prologue starts. We have split the function prologue and epilogue into its own methods for readability. This is not something that we have done before.

Let's look at the prologue, first. We save the frame pointer and the link registers, and we set our new frame pointer. We expect our parameters to be in the registers `r0`–`r3`. However, as soon as we start emitting the body of the function, we risk clobbering them, so we need to save them to the stack. Fortunately, ARM allows us to do it with a single `push` instruction. (An optimizing compiler would make an analysis called register allocation to determine which parameters can be kept in registers.)

Next, we emit the body of the function, which is followed by the function epilogue. Here we undo the effect of the function prologue: by setting the stack pointer value to frame pointer value we effectively deallocate whatever stack space we allocated (in the prologue, or otherwise). We set `r0` and thus our return value to `0`. This is to mimic the fact that JavaScript functions return `undefined` when there's no explicit return. We pick `0` because it is falsy, like `undefined`. This also removes the risk of returning a garbage value that was a leftover of some computation, in case we forget to have a `return` statement. As the last instruction, we restore the caller's frame pointer and pop the saved link register into the program counter, effectively returning from the function.

> **Explore…**
>
> We pushed four registers onto the stack. An improvement could be to specialize this code to handle the cases with fewer parameters more efficiently. If you do that, don't forget about 8-byte stack alignment.

So far, so good. Now, how do we access those parameters in the body of the function? They are located at well-known offsets from the frame pointer. Since stack grows from high addresses to low, we know that we can access parameters using negative offsets relative to the frame pointer. We also know that `push {r0, r1, r2, r3}` pushes the registers in the reverse order, same as this equivalent (but bulkier) code:

```
push {r3}
push {r2}
push {r1}
push {r0}
```

That means that the fourth parameter (from register `r3`) will be stored at the location `fp - 4`, third (`r2`) at `fp - 8`, second (`r1`) at `fp - 12`, first at `fp - 16`.

So, when we encounter an identifier that refers to one of the parameters, all we need to know is the offset from the frame pointer to locate that parameter. Then we can load it with `ldr`. Here's an example of loading the first parameter:

```
ldr r0, [fp, #-16]
```

What we can do is to store the mapping from parameter name to an offset from frame pointer and make sure it is passed down to all emitters. That means that whenever we encounter an identifier node `Id` we can look up the offset in that mapping and then we will know how to load it. This mapping is usually called an *environment*.

We could just pass around a `Map<string, number>`, but my crystal ball tells me that we will need to extend this data structure. So, instead we will introduce a data class with this map as an instance variable called `locals`:

```
class Environment {
  /* Initial version */
  constructor(public locals: Map<string, number>) {}
}
```

We call this class `Environment`. Right now it only contains `locals` map, which is the environment for our local variables, but it could include other environments, such as global variable environment, or type environments, or an environment with function signatures. We called our map `locals` and not `parameters`, because we foresee that we will use it for other local variables, and not just for parameters. It often grows to hold much more, and then a good name for it could be a more general `Context`.

In the `Environment` constructor, we make the `locals` parameter optional with the default value being an empty map. This is a mutable map, which we will modify. Fortunately, JavaScript default values are evaluated on each call so that we will get a new mutable map each time. But this is not the case in some of the other languages.

Now, there is a hefty change that we need to make. We need to adjust the `AST` interface to take the new `Environment` parameter,

which we will pass to `emit`. We will consistently call this parameter `env`.

```
interface AST {
  emit(Environment): void;
  equals(AST): boolean;
}
```

This will also require us to modify all AST nodes to take this environment and pass it down to all other node emitters. This is a very invasive, but mechanical change. We'll not go through all nodes for this, let's just use `Add` as our example:

```
class Add implements AST {
  constructor(public left: AST,
              public right: AST) {}

  emit(env: Environment) {
    this.left.emit(env);
    emit(`  push {r0, ip}`);
    this.right.emit(env);
    emit(`  pop {r1, ip}`);
    emit(`  add r0, r1, r0`);
  }

  equals(other: AST) {…}
}
```

The emitter now takes an `env` parameter, which it passes down to both its child AST nodes: `left` and `right`, in this case.

Now, back to our `Function`. We need to modify it to fulfil the new `AST` interface with `Environment`. However, we will ignore the incoming environment since we do not support nested functions here. We will bind it to an underscore to signify to the reader of this code that this variable is not used on purpose. Here's our new take on function definition:

```
class Function implements AST {
  constructor(…) {}

  /* Second draft */
  emit(_: Environment) {
    if (this.parameters.length > 4)
      throw Error(
```

```
          "More than 4 params is not supported");

      emit(``);
      emit(`.global ${this.name}`);
      emit(`${this.name}:`);
      this.emitPrologue();
      let env = this.setUpEnvironment();
      this.body.emit(env);
      this.emitEpilogue();
    }

    setUpEnvironment() {
      let locals = new Map();
      this.parameters.forEach((parameter, i) => {
        locals.set(parameter, 4 * i - 16);
      });
      return new Environment(locals);
    }

    …
  }
```

Here, in the function definition node, we know both the names of the parameters and the offsets at which they will be stored. So this is the ideal place to create a new environment. We do that in a method called `setUpEnvironment` which we call from `emit`. First, we create an empty environment, and then we loop through the parameters using `forEach` and set the locals map for each parameter to its offset relative to the frame pointer. We use the fact that `forEach` passes not only each item but also (optionally) each item's index `i`. To convert each index to its frame pointer offset we multiply the index by four to convert from words to bytes, and remove `16` since this is how many bytes we had allocated on the stack when we popped the four parameters in the prologue. We pass the constructed environment when we emit the body. It will, in turn, pass it to all its children, and so forth.

Now, a node that needs to access a local parameter can look it up in the environment. And what node could it be if not `Id`, the node for the identifiers in our code? All the hard work is done in the `Function` node, so our `Id` node will be quite simple:

```
  class Id implements AST {
```

```
  constructor(public value: string) {}

  emit(env: Environment) {
    let offset = env.locals.get(this.value);
    if (offset) {
      emit(`  ldr r0, [fp, #${offset}]`);
    } else {
      throw Error(
        `Undefined variable: ${this.value}`);
    }
  }

  equals() {…}
}
```

It looks up the offset from the local environment by identifier name. If the name is not in the environment, it throws an error. This means that a variable was used before it is defined. Otherwise, it loads a value using `ldr` relative to the frame pointer.

These frame pointers are handy, aren't they?

Now that we have implemented functions and parameters, we can implement quite a few interesting ones! First, we can get rid of the `Main` node, and generate the `main` function just like any other. Second, we can get rid of our `Assert` primitive, and implement it instead as a function:

```
function assert(x) {
  if (x) {
    putchar(46);
  } else {
    putchar(70);
  }
}
```

Finally, it's good to test that we got parameter passing right. Here's one of such tests:

```
function assert1234(a, b, c, d) {
  assert(a == 1);
  assert(b == 2);
  assert(c == 3);
  assert(d == 4);
}
```

It could be called as `assert1234(1, 2, 3, 4)` from `main`.

It feels like we can implement almost anything now; however, one critical piece is still missing.

## 8.11 Return

Functions must be able to return values. Even fans of continuation-passing style would agree. And thanks to frame pointers, implementing the `Return` node is easy:

```
class Return implements AST {
  constructor(public term: AST) {}

  emit(env) {
    this.term.emit(env);
    emit(`  mov sp, fp`);
    emit(`  pop {fp, pc}`);
  }

  equals(other: AST) {…}
}
```

We only need to emit the node (which is returned) into `r0`, and then repeat the same instructions as in the epilogue.

Now, the language of our compiler is technically-speaking Turing-complete. Loops, you say? We can loop using recursion, which we get for free since our functions are using the stack. (We are not getting tail-call optimization for free, though). This step warrants extensive testing, but the one test I have in mind is this:

```
function factorial(n) {
  if (n == 0) {
    return 1;
  } else {
    return n * factorial(n - 1);
  }
}
```

The factorial function! We can call it from `main` as follows:

```
assert(factorial(5) == 5);
```

It's a small dot for a test suite, but a giant leap for our compiler!

You've got all the essentials in place now. We will finish this chapter with a couple of niceties: local variables, assignments, and while loops. As a treat.

## 8.12 Local variables

Here we will set out to implement local variables, as declared with `var` keyword in JavaScript and represented as `Var` node in our AST.

Why `var` and not `let`? We have used only `let` in the implementation of this compiler!

Let's remind ourselves of the difference. Here's a function using `let`:

```
function f() {
  let x = 1;
  {
    let x = 2;
  }
  console.log(x);
}
```

And the same one using `var`:

```
function f() {
  var x = 1;
  {
    var x = 2;
  }
  console.log(x);
}
```

The first one prints `1`, and the second one prints `2`. The difference is that the `var` bindings work at the function scope, and `let` bindings work at the block scope. So the two "vars" refer to the same variables, but the two "lets" are different: the second one is within another scope delimited by braces.

The reason we implement `var` is that it simplifies scope handling: you only need one scope per function. That's probably also the reason JavaScript introduced `var` first, and `let` much later. There's

also `const`, which is just like `let`, but makes the assignment forbidden on its bindings.

So, how do we implement `var`? We can push the value on the stack, then look it up in the environment, just like with parameters. However, when defining parameters, we know their offset: they are at the beginning of the frame. However, when we emit a `var`, we don't know how far down the stack we currently are. But we can track this in the environment!

Let's modify the `Environment` class to store the `nextLocalOffset` number, which represents the next available frame pointer offset. This is also sometimes called a *stack index*.

```
class Environment {
  constructor(public locals: Map<string, number>,
              public nextLocalOffset: number) {}
}
```

By default, we initialize to `0`. However, in `Function` we need to set it up depending on how many parameters we allocate. Right now we always allocate four, at offsets `-4`, `-8`, `-12`, and `-16`. So the next available offset is `-20`. And that's the value that we use in the environment setup:

```
class Function implements AST {
  …
  setUpEnvironment() {
    let locals = new Map();
    this.parameters.forEach((parameter, i) => {
      locals.set(parameter, 4 * i - 16);
    });
    nextLocalOffset = -20;
    return new Environment(locals, nextLocalOffset);
  }
  …
}
```

Now, onto the `Var` node. Theoretically, we could add the variable name to the `local` environment mapping, so it maps to the `nextLocalOffset` value, then push the value onto the stack, and update the `nextLocalOffset` to point to the next available offset. However, we need to maintain 8-byte alignment, so it is slightly more involved:

```
class Var implements AST {
  constructor(public name: string,
              public value: AST) {}

  emit(env: Environment) {
    this.value.emit(env);
    emit(`  push {r0, ip}`);
    env.locals.set(this.name, env.nextLocalOffset - 4);
    env.nextLocalOffset -= 8;
  }

  equals(other: AST) {…}
}
```

We still start by emitting the value and pushing it onto the stack, but we also need to push something like an `ip` register to keep the stack 8-byte aligned. This way it's the `ip` that will be located at the `nextLocalOffset`, so when updating the local environment, we subtract another 4 bytes from it. Now, we need to advance `nextLocalOffset` two stack slots ahead; in other words, we decrement it by 8.

> **Explore…**
>
> This wastes 50% of the stack space for locals. Here's a way to fix it: you can add an array of offsets `vacantOffsets` to the `Environment`. Then each time you need a stack slot you first check if there are any vacant slots, and try to use it (and remove it from the array). And only if there are no such slots, you allocate new stack space. This technique can also be used to take advantage of over-allocating in other situations, for example, when you have an odd number of function parameters. There's also a way to organize it all neatly into a `Frame` data structure that handles this offset twiddling in one place, so every relevant emitter doesn't have to keep track of the offset math.

Let's add an assertion to test our `var` handling:

```
var x = 4 + 2 * (12 - 2);
var y = 3 * (5 + 1);
var z = x + y;
assert(z == 42);
```

## 8.13 Assignment

Assignment handling is very similar to identifier look-up. Except instead of reading its value, we are writing it. We emit the value into `r0`. Then we look up the frame pointer offset in the local environment. If the environment look-up fails, we throw an error: that variable was not defined at this point of time. Otherwise, we use the `str` instruction to store the value from `r0` into the frame pointer offset.

```
class Assign implements AST {
  constructor(public name: string,
              public value: AST) {}

  emit(env: Environment) {
    this.value.emit(env);
    let offset = env.locals.get(this.name);
    if (offset) {
      emit(`  str r0, [fp, #${offset}]`);
    } else {
      throw Error(`Undefined variable: ${this.name}`);
    }
  }

  equals(other: AST) {…}
}
```

A simple test for the assignment:

```
var a = 1;
assert(a == 1);
a = 0;
assert(a == 0);
```

**Explore…**

Implement `const` bindings similar to `var`, but such that assignment is not allowed on them. A way to do that would be to change the locals mapping to include not only an offset but also a flag that signifies whether the binding is constant or not. Then, when the assignment looks it up, check that the constant flag is not set, and fail otherwise.

## 8.14 While-loop

While-loop handling is in many ways similar to the `If` statement handling. It has a conditional expression, which is checked for truthiness, but it has only one branch. The other difference is that at the end of that branch, it jumps back to the beginning:

```
class While implements AST {
  constructor(public conditional: AST,
              public body: AST) {}

  emit(env: Environment) {
    let loopStart = new Label();
    let loopEnd = new Label();

    emit(`${loopStart}:`);
    this.conditional.emit(env);
    emit(`  cmp r0, #0`);
    emit(`  beq ${loopEnd}`);
    this.body.emit(env);
    emit(`  b ${loopStart}`);
    emit(`${loopEnd}:`);
  }

  equals(other: AST) {…}
}
```

Here we generate two unique labels: `loopStart` and `loopEnd`. We check the conditional and branch to `loopEnd` if it is falsy. Then follows the body of the loop. In the end, we unconditionally branch to the `loopStart` label.

A quick test for the `while` loop may look like this:

```
var i = 0;
while (i != 3) {
  i = i + 1;
}
assert(i == 3);
```

We've put away the while loop handling for so long because to test it we first need the assignment to work.

We can now also implement a version of `factorial` function using `while` loop:

```
function factorial(n) {
  var result = 1;
  while (n != 1) {
    result = result * n;
    n = n - 1;
  }
  return result;
}
```

**Explore…**

A `for` loop is a special case of the `while` loop. Consider that the following `for` loop:

```
for (i = 1; i < 5; i = i + 1) {
  body();
}
```

Is equivalent to this `while` loop:

```
i = 1;
while (i < 5) {
  i = i + 1;
  body();
}
```

# Part II

# Compiler Extensions

# Chapter 9

# Introduction to Part II

Before extending the compiler, let's discuss the language we have implemented so far.

Is our language memory-safe? What is memory safety, anyway? Simply speaking, a language is memory-safe if it does not allow you to write a program that causes a segmentation fault. The baseline language is memory-safe if we limit ourselves to calling functions that we have defined ourselves. However, our calling convention allows us to call arbitrary `libc` functions. You can find creative ways to call these functions that will lead to a segmentation fault (try `free(42)`). So, unless we do something about that, the baseline language is not memory-safe.

A way to fix that is to introduce a prefix for function labels. For example, a function `factorial` can be compiled with a label `ts$factorial:`, and a call to `factorial` can be compiled to a jump to `ts$factorial`. This way, you can only call functions that are defined in the source language, or that had explicit wrappers written in assembly. These wrappers can be auto-generated by the compiler and also include type conversion, if necessary.

Is our language dynamically-typed? Or is it statically-typed? Both and neither! The baseline language supports only integer numbers. So, it could be thought of as a dynamically-typed language with only one data type, or as a statically-typed language with one static type. But we are soon to change this.

However, before we explore static and dynamic typing, we need

to have more than one data type in our language. We will start by introducing booleans, undefined, and then arrays. First, we will introduce them in an unsafe/untyped manner, and then we will apply static/dynamic treatment to them.

# Chapter 10

# Primitive Scalar Data Types

When we say *scalar data types* we mean data types that fit into a single machine word, which is not a pointer but carries a value in itself. We already have one scalar data type for integer numbers.

Let's introduce a boolean data type. First, we need an AST node for it:

```
class Boolean implements AST {
  constructor(public value: boolean) {}

  emit(env: Environment) {
    if (this.value) {
      emit(`  mov r0, #1`);
    } else {
      emit(`  mov r0, #0`);
    }
  }

  equals(other: AST): boolean {…}
}
```

We emit it the same way as integers 1 and 0 (for `true` and `false`).

In the parser, we introduce new tokens for `true` and `false`, and compose them to create a `boolean` parser:

```
let TRUE =
  token(/true\b/y).map((_) => new Boolean(true));
let FALSE =
  token(/false\b/y).map((_) => new Boolean(false));

let boolean: Parser<AST> = TRUE.or(FALSE)
```

And we can extend the `atom` rule of the expression grammar by adding a `boolean` alternative. A good idea at this point is to introduce an additional `scalar` rule:

```
scalar <- boolean / ID / NUMBER
atom <-
  call / scalar / LEFT_PAREN expression RIGHT_PAREN
```

Then, after implementing this grammar as a parser we get booleans in our extended baseline language:

```
assert(true);
assert(!false);
```

However, they behave exactly like integers `1` and `0`, so `assert(true == 1)` will succeed. Under static typing, this comparison would be rejected by the compiler. Under dynamic typing, this would evaluate to `false` at run-time.

---

Similarly, we can add other scalars, such as `undefined`, `null` (that compile to 0, like `false`), or a character type, which could compile to the integer value of its ASCII code (though, JavaScript and TypeScript treat characters as strings).

Table 10.1: Summary of AST constructor signatures with examples

| AST Constructor Signature | Example |
|---|---|
| `Boolean(value: boolean)` | **`true`** |
| `Undefined()` | **`undefined`** |
| `Null()` | **`null`** |

# Chapter 11

# Arrays and Heap Allocation

Let's implement simple arrays, which we can create using literal notation like `[10, 20, 30]`, extract an element by indexing (`a[0]`) and querying for their length (`length(a)`).

```
let a = [10, 20, 30];
assert(a[0] == 10);
assert(a[1] == 20);
assert(a[2] == 30);
assert(a[3] == undefined);  // Bounds checking.
assert(length(a) == 3);
```

We would also like to implements bounds checking. In other words, if we ask for an index that is out of bounds, we don't want a segmentation fault. In this case, in JavaScript, we expect `undefined` to be returned. In other languages, it could be `null` or a raised exception. We ignore the fact that `undefined` is nothing but a glorified `0` until later.

To cover this functionality, we need three new AST nodes, listed in the following table.

Table 11.1: Summary of AST constructor signatures with examples

| AST Constructor Signature | Example |
|---|---|
| `ArrayLiteral(args: Array<AST>)` | `[a1, a2, a3]` |
| `ArrayLookup(array: AST, index: AST)` | `array[index]` |
| `Length(array: AST)` | `length(array)` |

We have picked the name `ArrayLiteral` as opposed to `Array` in order not to clash with JavaScript built-in `Array` class that we use a lot. We decided to use a `length` function as opposed to a method (like in JavaScript) since we don't have support for methods yet. However, we could have a special syntax for `x.length` just for this purpose.

We add two new tokens: `LEFT_BRACKET` and `RIGHT_BRACKET` standing for "`[`" and "`]`". We also extend our grammar (and parser) with two new rules that we integrate into the `atom` rule:

```
arrayLiteral <- LEFT_BRACKET args RIGHT_BRACKET
arrayLookup <-
 ID LEFT_BRACKET expression RIGHT_BRACKET
atom <-
 call / arrayLiteral / arrayLookup / scalar
 / LEFT_PAREN expression RIGHT_PAREN
```

There are several ways to lay out an array in memory. The simplest is the layout of a fixed-size array. Such array is represented as a pointer to a single stretch of memory containing array length and the elements. In the following figure, you can see such array layout for array `a` from our previous example.

However, fixed-size arrays are, well, fixed-size. To implement growable or resizable arrays, we need a more sophisticated layout, like the one in the figure after that. It stores array length and its *capacity*, as well as a pointer to the actual array of elements. The idea is to *over-allocate* array to some extent, such that the capacity is larger than the length. This way, elements could be added to the array without memory allocation (to some extent). When an array grows beyond capacity, a new span of memory is allocated for the elements.

Let's look at code generation for fixed-size arrays.

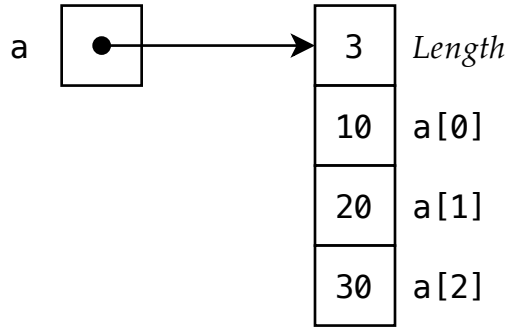


Figure 11.1: Word-diagram of a fixed-sized array layout

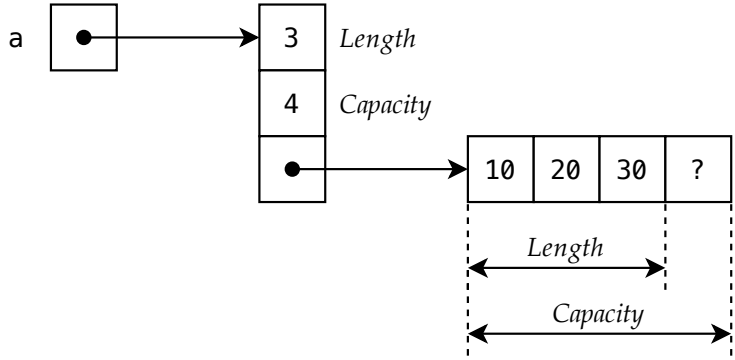


Figure 11.2: Word-diagram of a resizable array layout

## 11.1 Array literals

We start with code generation for array literals.

```
class ArrayLiteral implements AST {
  constructor(elements: Array<AST>) {}

  emit(env: Environment) {
    let length = this.elements.length;
    emit(`  ldr r0, =${4 * (length + 1)}`);
    emit(`  bl malloc`);
    emit(`  push {r4, ip}`);
    emit(`  mov r4, r0`);
    emit(`  ldr r0, =${length}`);
    emit(`  str r0, [r4]`);
    this.elements.forEach((element, i) => {
      element.emit(env);
      emit(`  str r0, [r4, #${4 * (i + 1)}]`);
    });
    emit(`  mov r0, r4`);
    emit(`  pop {r4, ip}`);
  }

  equals(other: AST): boolean {…}
}
```

First, we call `malloc` to allocate enough memory to store the length of the array and the elements. Since `malloc` takes the number of *bytes* to be allocated we need to multiply the length by four and add one more word to store the length itself.

```
    let length = this.elements.length;
    emit(`  ldr r0, =${4 * (length + 1)}`);
    emit(`  bl malloc`);
```

Then, `malloc` returns a pointer to the freshly-allocated memory in `r0`. However, `r0` is a lousy register for this. As we emit code for each array element, they might have calls that clobber `r0`. Thus, let's use `r4` for this, which is a call-preserved register. But before we do that we need to save the previous value of `r4` on the stack.

```
    emit(`  push {r4, ip}`);
    emit(`  mov r4, r0`);
```

Next, we store array length in the first word of the allocated span

of memory.

```
emit(`  ldr r0, =${length}`);
emit(`  str r0, [r4]`);
```

After that, we emit code for each element and store it into the corresponding memory slot.

```
this.elements.forEach((element, i) => {
  element.emit(env);
  emit(`  str r0, [r4, #${4 * (i + 1)}]`);
});
```

We finish by returning the pointer in `r0` and restoring the call-preserved `r4`.

```
emit(`  mov r0, r4`);
emit(`  pop {r4, ip}`);
```

## 11.2 Array lookup

Next is code generation for an array lookup.

```
class ArrayLookup implements AST {
  constructor(array: AST, index: AST) {}

  emit(env: Environment) {
    this.array.emit(env);
    emit(`  push {r0, ip}`);
    this.index.emit(env);
    emit(`  pop {r1, ip}`);
    emit(`  ldr r2, [r1]`);
    emit(`  cmp r0, r2`);
    emit(`  movhs r0, #0`);
    emit(`  addlo r1, r1, #4`);
    emit(`  lsllo r0, r0, #2`);
    emit(`  ldrlo r0, [r1, r0]`);
  }

  equals(others: AST) {…}
}
```

First, we store the array pointer in `r1` and array index in `r0`.

```
    this.array.emit(env);
    emit(`  push {r0, ip}`);
    this.index.emit(env);
    emit(`  pop {r1, ip}`);
```

Then, we load array length into `r2` and compare it with the array index to perform bounds checking. If array index is out of bounds, we do a conditional `mov` with `hs` suffix and return zero. If it succeeds, then we execute three instructions with `lo` suffix. In these three instructions, we convert array index into byte offset. We add 4 to skip over the length slot, and we do *logical shift left* or `lsl` to convert from word offset to byte offset. Shifting left by two is virtually the same as multiplying by 4.

```
    emit(`  ldr r2, [r1]`);
    emit(`  cmp r0, r2`);
    emit(`  movhs r0, #0`);
    emit(`  addlo r1, r1, #4`);
    emit(`  lsllo r0, r0, #2`);
    emit(`  ldrlo r0, [r1, r0]`);
```

However, using some of the ARM assembly language features that we have not covered in the book (like auto-increment and barrel shifter) we can shortern the same code to this:

```
    emit(`  ldr r2, [r1], #4`);
    emit(`  cmp r0, r2`);
    emit(`  movhs r0, #0`);
    emit(`  ldrlo r0, [r1, r0, lsl #2]`);
```

Which is an elegant way to do a bounds check and convert array index into a byte offset at the same time.

## 11.3 Array length

Array length can be obtained by following as the slot directly pointed by the array.

```
class Length implements AST {
  constructor(array: AST) {}

  emit(env: Environment) {
    this.array.emit(env);
    emit(`  ldr r0, [r0, #0]`);
```

```
    }

    equals(others: AST) {…}
  }
```

## 11.4 Strings

Strings can be thought as an array of bytes with a particular encoding. They can be implemented similarly to arrays, but using byte variants of the load and store instructions: `ldrb` and `strb` instead of the regular `ldr` and `str`. However, when we see `a[i]` we need to know if it's an array or a string to execute the right code. We either need to know the static type of `a` at compile-type, or its dynamic type (or tag) by inspecting the data-structure at run-time. But before we jump into that, we need to learn about one particular pattern that will help us maintain our code.

# Chapter 12

# Visitor Pattern

We are about to add more passes to our compiler: type checking and code generation for dynamic typing. What we could do is extend the AST interface with new methods, one for each pass. It can look something like this:

```
interface AST {
  emit(env: Environment): void;
  emitDynamic(env: Environment): void;
  typeCheck(env: TypeEnvironment): Type;
  equal(other: AST): boolean;
}
```

And this is perfectly fine. However, this way code for each pass is intertwined with code for other passes. In other words, code is grouped by AST node and not by pass.

However, using the *visitor pattern* we can group the code for each pass together under a separate class. The visitor pattern allows us to decouple our passes from AST by using indirection. Instead of having a method *per pass, per AST node* we add a single method *per AST node* called `visit` that delegates the action to a class that implements the *visitor interface*. The *visitor interface* has one method per AST node: `visitAssert`, `visitLength`, `visitNumber`, etc.

```
interface AST {
  visit<T>(v: Visitor<T>): T;
  equal(other: AST): boolean;
}
```

```
interface Visitor<T> {
  visitAssert(node: Assert): T;
  visitLength(node: Length): T;
  visitNumber(node: Number): T;
  visitBoolean(node: Boolean): T;
  visitNot(node: Not): T;
  visitEqual(node: Equal): T;
  …
}
```

Each AST node implements the new `AST` interface by calling the corresponding visitor method. For example, `Assert` calls `visitAssert`, `Length` calls `visitLength`, etc.

```
class Assert implements AST {
  constructor(public condition: AST) {}

  visit<T>(v: Visitor<T>) {
    return v.visitAssert(this);
  }

  equals(other: AST): boolean {…}
}

class Length implements AST {
  constructor(public array: AST) {}

  visit<T>(v: Visitor<T>) {
    return v.visitLength(this);
  }

  equals(other: AST): boolean {…}
}
```

The visitor interface `Visitor<T>` is generic. That means it can be used to implement passes that return different things. For example, `Visitor<AST>` produces an `AST` node, `Visitor<void>` can emit code as a side-effect.

Let's convert our existing code generation pass into a visitor. Since our existing `emit` method returned `void`, our new visitor will implement `Visitor<void>`. Instead of having a separate `Environment` class, we make the visitor constructor take all the environ-

ment parameters. In a way, the visitor becomes the environment.

```
class CodeGenerator implements Visitor<void> {
  constructor(public locals: Map<string, number> = new Map(),
              public nextLocalOffset: number = 0) {}

  visitAssert(node: Assert) {
    node.condition.visit(this);
    emit(`  cmp r0, #1`);
    emit(`  moveq r0, #'.'`);
    emit(`  movne r0, #'F'`);
    emit(`  bl putchar`);
  }

  visitLength(node: Length) {
    node.array.visit(this);
    emit(`  ldr r0, [r0, #0]`);
  }

  …
}
```

We copy the body of each method, like `Assert.emit` and `Length.emit` into the visitor methods `visitAssert` and `visitLength`.

In `emit` methods we used to call `emit` recursively for inner nodes, like this:

```
emit(env: Environment): void {
  this.array.emit(env);
  emit(`  ldr r0, [r0, #0]`);
}
```

Now, instead, we call the `visit` method on them.

```
visitLength(node: Length) {
  node.array.visit(this);
  emit(`  ldr r0, [r0, #0]`);
}
```

Previously `this` referred to the AST node, but now the node is passed as the parameter called `node`. Now, `this` refers to the visitor itself, which we pass instead of the `env` parameter.

In rare places where we created a new environment, we create a

new visitor instead with the updated environment. Here's an example from `visitFunction`.

Before:

```
let env = new Environment(locals, -20);
this.body.emit(env);
```

After:

```
let visitor = new CodeGenerator(locals, -20);
node.body.visit(visitor);
```

As you can see, converting from an AST-based pass to a visitor-based pass is a purely mechanical transformation. The next passes that we will introduce will also be based on the visitor pattern.

> **Note:**
>
> If you are using a functional programming language you might notice that the visitor pattern corresponds to pattern matching.

# Chapter 13

# Static Type Checking and Inference

In this chapter, we will implement a type checker for our language with *local type inference*. Local type inference means that local variables do not need a type annotation. So you can write the following:

```
let a = [1, 2, 3];
```

And the type of `a` will be inferred as `Array<number>`.

Our type checker will cover such types as `number`, `boolean`, `void`, and `Array<T>` where `T` could be any other type, for example `Array<Array<boolean>>`. We will also deal with function types, such as `(x: boolean, y: number) => number`, when type checking function calls.

First, we need to refer to different types to manipulate them. We will represent them similarly to AST nodes, with an interface called `Type` and one class that implements it per type.

```
interface Type {
  equals(other: Type): boolean;
  toString(): string;
}
```

We will need equality and a `toString` method to be able to display type errors.

Table 13.1: Summary of Type constructor signatures with examples

| Type Constructor Signature | Example |
|---|---|
| `BooleanType()` | `boolean` |
| `NumberType()` | `number` |
| `VoidType()` | **`void`** |
| `ArrayType(element: Type)` | `Array<number>` |
| `FunctionType(`<br>`  parameters: Map<string, Type>,`<br>`  returnType:Type,`<br>`)` | `(x: boolean, y: number)`<br>`  => number` |

Implementing `toString` and `equals` for these data classes is straightforward. However, `FunctionType` requires some comments:

- First, `Map` in JavaScript preserves the order of elements, which is important to be able to match parameter positions in a call site.
- When comparing two instances of `FunctionType` it makes sence to ignore parameter names, since `(x: boolean, y: number) => number` is the same type as `(p1: boolean, p2: number) => number`.

Our language will change to allow functions to be type-annotated. We will first change our `Function` AST node to store its signature as `FunctionType`, instead of just parameter names:

```
class Function implements AST {
  constructor(public name: string,
              public signature: FunctionType,
              public body: AST) {}

  visit<T>(v: Visitor<T>) {…}

  equals(other: AST): boolean {…}
```

```
}
```

We will also need to change our grammar and parser. After introducing the necessary tokens, we can define the following grammar for types.

```
arrayType <- ARRAY LESS_THAN type GREATER_THAN
type <- VOID | BOOLEAN | NUMBER | arrayType
```

The `type` rule is recursive, just like `expression` and `statement`.

Next, we need to change the grammar and parser for functions, to include optional type annotations:

```
optionalTypeAnnotation <- (COLON type)?
parameter <- ID optionalTypeAnnotation
parameters <- (parameter (COMMA parameter)*)?
functionStatement <-
  FUNCTION ID LEFT_PAREN
    parameters
  RIGHT_PAREN optionalTypeAnnotation
    blockStatement
```

We allow optional type annotations for function parameters and function return type. Why optional? Some languages do infer parameter and return types, but we do it so that we can use the same parser unmodified later with dynamic types. If a type annotation is missing we default it to `number`, which also gives us some backwards compatibility with our test suite.

Now, the parser will accept functions with type annotations like the following.

```
function pair(x: number, y: number): Array<number> {
  return [x, y];
}
```

Fur such function, the parser will produce a `Function` node with `FunctionType` signature like this:

```
new Function(
  "pair",
  new FunctionType(
    new Map([["x", new NumberType()], ["y", new NumberType()]]),
    new ArrayType(new NumberType()),
  ),
  new Block([
```

```
      new Return(new ArrayLiteral([new Id("x"), new Id("y")])),
    ]),
  )
```

Next, for type checking, we will use a function that asserts that two types are the same and otherwise raises an exception to signal an error. We call it `assertType`:

```
function assertType(expected: Type, got: Type): void {
  if (!expected.equals(got)) {
    throw TypeError(
      `Expected ${expected}, but got ${got}`);
  }
}
```

It uses both the `equals` method and the `toString` method, which is invoked implicitly for template string variables. Here, we are slightly abusing the built-in `TypeError` exception, which has a different purpose (run-time type errors), so it is better to define a custom exception type.

Our `TypeChecker` pass will walk the AST and will either abort with a `TypeError`, or will return the inferred `Type` of an expression. Thus, `TypeChecker` implements `Visitor<Type>`.

```
class TypeChecker implements Visitor<Type> {
  constructor(
    public locals: Map<string, Type>,
    public functions: Map<string, FunctionType>,
    public currentFunctionReturnType: Type | null,
  ) {}

  …
}
```

It maintains two environments:

- `locals`—an environment that stores types of local variables in a function, and
- `functions`—an environment that stores signatures of each function.

We need two separate environments for these since functions are not first-class in our language: they cannot be assigned to a variable and passed around. So, when we encounter a `Call`, we will search for a function in the `functions` map, and when

we encounter an `Id`, we will search for a non-function type in a `locals` map.

We also maintain an instance variable `currentFunctionReturnType` which will help us type-check `Return` statements.

Now we get to the actual type checking and inference.

## 13.1 Scalars

The type of literal scalars is the easiest to infer. We know that the type of a number literal is `number`, a type of boolean node is `boolean`, and so forth.

```
visitNumber(node: Number) {
  return new NumberType();
}

visitBoolean(node: Boolean) {
  return new BooleanType();
}
```

In TypeScript the `void` type refers to expressions or statements that return `undefined`.

```
visitUndefined(node: Undefined) {
  return new VoidType();
}
```

## 13.2 Operators

Now, let's look at the most straightforward operator—negation. That's not exactly how it works in TypeScript, but let's assume that the negation operator expects strictly boolean parameter. To enforce that we first infer the inner term's type by calling the `visit` method on it. Then we assert that it is boolean:

```
visitNot(node: Not) {
  assertType(new BooleanType(), node.term.visit(this));
  return new BooleanType();
}
```

We return the `boolean` type since this will always be the result of negation.

To type-check numeric operators like `Add`, we infer the type of `left` and `right` parameters and assert that they are both numbers. Then we return `number` as the resulting type.

```
visitAdd(node: Add) {
  assertType(new NumberType(), node.left.visit(this));
  assertType(new NumberType(), node.right.visit(this));
  return new NumberType();
}
```

Now we get to something more interesting. Operations such as equality are generic: they can work with any type as long as the type on the left-hand side is the same as the one on the right-hand side. We infer the types on both sides by visiting them, and then assert that the two are the same:

```
visitEqual(node: Equal) {
  let leftType = node.left.visit(this);
  let rightType = node.right.visit(this);
  assertType(leftType, rightType);
  return new BooleanType();
}
```

## 13.3 Variables

When we encounter a new variable defined with `var`, we infer its type by visiting it, and then we add that type to the `locals` environment.

```
visitVar(node: Var) {
  let type = node.value.visit(this);
  this.locals.set(node.name, type);
  return new VoidType();
}
```

When we encounter a variable, we look it up in the `locals` environment and return its type. If the variable is not defined—we raise a `TypeError`, which makes a similar check at the code generation level obsolete.

```
visitId(node: Id) {
  let type = this.locals.get(node.value);
  if (!type) {
    throw TypeError(
```

```
        `Undefined variable ${node.value}`);
    }
    return type;
  }
```

When assigning a new value to a variable, we check two things:

- First, that the variable is previously defined.
- Second, that the assignment does not change the type of the variable.

```
visitAssign(node: Assign) {
  let variableType = this.locals.get(node.name);
  if (!variableType) {
    throw TypeError(
      `Assignment to an undefined variable ${node.name}`);
  }
  let valueType = node.value.visit(this);
  assertType(variableType, valueType);
  return new VoidType();
}
```

## 13.4 Arrays

Inferring the type of array literals is a bit trickier than the other literals that we've covered. We know that any array literal is of type `Array<T>`, but we need to figure out what `T` is. First of all, we can't infer the type of an empty array—there's simply not enough information at this point. There are bi-directional type inference algorithms that handle this, but usually, this is solved by requiring a type annotation.

If the array is non-empty, we infer the type of each element and then assert pair-wise that they are the same type.

```
visitArrayLiteral(node: ArrayLiteral): Type {
  if (node.args.length == 0) {
    throw TypeError(
      "Can't infer type of an empty array");
  }
  let argsTypes =
    node.args.map((arg) => arg.visit(this));
  // Assert all arguments have the same type, pair-wise
```

```
    let elementType = argsTypes.reduce((prev, next) => {
      assertType(prev, next);
      return prev;
    });
    return new ArrayType(elementType);
  }
```

For something like our array length primitive, we need to assert that the parameter type is an array, but we don't care about the array element type. So, instead of using `assertType` we manually check that the type is an instance of `ArrayType`, and raise a `TypeError` otherwise. The inferred type of such expression is `number`.

```
  visitLength(node: Length) {
    let type = node.array.visit(this);
    if (type instanceof ArrayType) {
      return new NumberType();
    } else {
      throw TypeError(
        `Expected an array, but got ${type}`);
    }
  }
```

When handling array lookup, we assert that the index is a number, and that the array is of type array:

```
  visitArrayLookup(node: ArrayLookup): Type {
    assertType(new NumberType(), node.index.visit(this));
    let type = node.array.visit(this);
    if (type instanceof ArrayType) {
      return type.element;
    } else {
      throw TypeError(
        `Expected an array, but got ${type}`);
    }
  }
```

## 13.5 Functions

When encountering a function, we add its signature to the `functions` environment. We do not need to infer it, because it is parsed from the source. Before type-checking the body of the function, we create a new visitor with a modified environment. Since we

use mutable maps, we need to pass a new `Map` to each function, to avoid modifying the wrong function's environment. We also set the `currentFunctionReturnType` to the one in the signature.

```
visitFunction(node: Function) {
  this.functions.set(node.name, node.signature);
  let visitor = new TypeChecker(
    new Map(node.signature.parameters),
    this.functions,
    node.signature.returnType,
  );
  node.body.visit(visitor);
  return new VoidType();
}
```

When visiting a `Call`, we fetch that function's signature from the `functions` environment. Then we infer the type of the function being called and compare it to the type from the environment. When inferring the type of the function we visit each argument, and since `FunctionType` requires a name for each parameter, we assign them dummy names, such as `x0`, `x1`, `x2`, etc. When it comes to the function's return type, we use the one from the type annotation.

```
visitCall(node: Call) {
  let expected = this.functions.get(node.callee);
  if (!expected) {
    throw TypeError(
      `Function ${node.callee} is not defined`);
  }
  let argsTypes = new Map();
  node.args.forEach((arg, i) =>
    argsTypes.set(`x${i}`, arg.visit(this))
  );
  let got =
    new FunctionType(argsTypes, expected.returnType);
  assertType(expected, got);
  return expected.returnType;
}
```

When checking a return statement, we infer the type of the returned value and compare it with the one from the function annotation, which we saved conveniently in `currentFunctionReturnType` instance variable.

```
visitReturn(node: Return) {
  let type = node.term.visit(this);
  if (this.currentFunctionReturnType) {
    assertType(this.currentFunctionReturnType, type);
    return new VoidType();
  } else {
    throw TypeError(
      "Encountered return statement outside any function");
  }
}
```

## 13.6 If and While

When checking `If` and `While` we visit each inner node to make sure that their type is checked, but we don't enforce any type. The `conditional` could be checked to be boolean, but in TypeScript (as in many languages) it is not required to be such. We could also enforce that the statements inside the branches return `void`, but this is not usually required either.

```
visitIf(node: If) {
  node.conditional.visit(this);
  node.consequence.visit(this);
  node.alternative.visit(this);
  return new VoidType();
}

 visitWhile(node: While) {
  node.conditional.visit(this);
  node.body.visit(this);
  return new VoidType();
}
```

However, if we had a ternary conditional operation, we would need to ensure that the two branches return the same type.

## 13.7 Error messages

We can improve our error messages by specializing them to each situation instead of relying on a generic message produced by `assertType`.

## 13.8 Soundness

Our type checker is complete. But is it *sound*? A type system is *sound* if the types at compile-time are always consistent with run-time. Can you find a soundness issue in our type checker?

Although we check that each `return` statement is consistent with the annotated return type, we don't check that each control-flow path has a return statement. Here's an example that demonstraits this issue:

```
function wrongReturnType(x: boolean): Array<number> {
  if (x) {
    return [42];
  }
}

function main() {
  var a = wrontReturnType(false);
  a[0]; // Segmentation fault
}
```

Checking that each control flow path leads to a return statement requires a bit of control-flow analysis which is outside of the scope of this book.

# Chapter 14

# Dynamic Typing

With dynamic typing, you expect to be able to query the data type of any variable, using `isinstance` or some other mechanism. Right now we can't distinguish at run-time between a number and (a pointer to) an array. With the static type system we have, it doesn't matter, because it disallows such comparison to be made at run-time.

So, how do we distinguish a pointer from a number, a false from zero, and so on? This is typically achieved with *tagging*.

## 14.1 Tagging

Tagging is reserving one or more bits in every word for a particular purpose, for example, to distinguish its type.

For our compiler, we will adopt a two-bit tagging scheme described in the following figure.

But how do we exactly "afford" these two bits? Doesn't it change the actual numeric or pointer value?

Let's talk about pointers first.

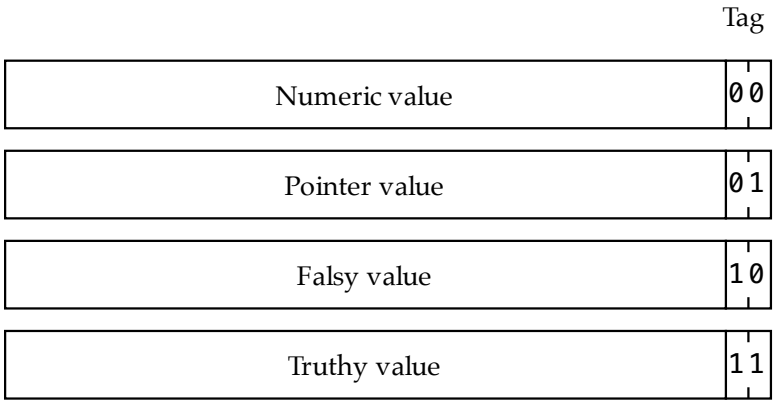


Figure 14.1: Our two-bit tagging scheme

## 14.2 Pointer tag

We expect pointers to be word-aligned. That means that the last two bits in a word will always be `0b00`. That means we can store a two-bit tag there to distinguish pointers.

The simplest way to deal with that is to select the tag `0b00`. And that's what many implementations do. But we can also select a different tag, like `0b01`, as we did. In the general case, when we want to operate on a pointer, we can remove the tag, do the operations, then put it back, if necessary. However, having a tag of `0b01` simply means that we point to the second byte in the word. So, to load the first word, instead of loading at offset `0`, we can load at offset `-1`. This way `ldr r0, [r1]` becomes `ldr r0, [r1, #-1]`, and `ldr r0, [r1, #4]` becomes `ldr r0, [r1, #3]` and so on! This means that not much is lost. At worst, it takes one instruction to clear the tag.

The only kind of pointers we handle right now are arrays. To distinguish them from other heap-allocated objects, it is common to encode information about the object type in the data structure itself.

Next, let's talk about integers.

## 14.3 Integer tag

If we dedicate two bits for tagging, that means that our integers shrink from 32-bit ones to 30-bit ones. That's an oddly-sized integer! However, in dynamically-typed languages, integers are often silently promoted to floating-point numbers or arbitrary-precision numbers. That means the fact that an integer is 30-bit is not exposed to the user of the language. These oddly-sized integers are called *small integers*.

So, of course, to deal with small integers we can remove the tags, perform an operation and then add the tag. However, that's not always necessary either. We have selected the tag `0b00` for integers. That means that most arithmetic and logic operations will work on such tagged integers as-is. For example, `3 + 4 = 7` used to be `0b11 + 0b100 = 0b111`, but now it's `0b11_00 + 0b100_00 = 0b111_00`. (We have used underscores to separate the tag and the value visually.) One notable exception to this is multiplication, which requires one instruction—shift right—to fix it up, and even then there are ways to incorporate that shift "for free" as a part of another instruction.

By selecting this tag, we can achieve decent performance and avoid changing our code generation pass too much.

## 14.4 Truthy and falsy tags

We have two more tags available: `0b10` and `0b11`. And we use them for all the falsy and truthy values, correspondingly. This is to highlight the fact that tags are often selected in creative ways to simplify common operations. We only have `true`, `false`, and `undefined` left in our language. We will assign them values `0b1_11`, `0b1_10`, and `0b0_10`, correspondingly. This also allows us to quickly check if something is boolean by checking that the bits `0b1_10` are set.

## 14.5 Code generation

Now, we will modify our code generation pass to adapt to our new tagging strategy. But first, let's introduce some useful constants. We start with the four tags:

```
let numberTag = 0b00;
let pointerTag = 0b01;
let falsyTag = 0b10;
let truthyTag = 0b11;
```

Then the bit patterns of our literal values `true`, `false`, and `undefined`:

```
let trueBitPattern = 0b111;
let falseBitPattern = 0b110;
let undefinedBitPattern = 0b010;
```

And a helper function to convert an integer to a small integer (in other words, to add tag to an integer):

```
let toSmallInteger = (n: number) => n << 2;
```

And finally, a bitmask that will help us extract the tag from a word:

```
let tagBitMask = 0b11;
```

Now, onto the code generation pass.

## 14.6 Literals

Generating an integer number now uses the `toSmallInteger` helper:

```
visitNumber(node: Number) {
  emit(`  ldr r0, =${toSmallInteger(node.value)}`);
}
```

Boolean and undefined values use the defined constants:

```
visitBoolean(node: Boolean) {
  if (node.value) {
    emit(`  mov r0, #${trueBitPattern}`);
  } else {
    emit(`  mov r0, #${falseBitPattern}`);
  }
}

visitUndefined(node: Undefined) {
  emit(`  mov r0, #${undefinedBitPattern}`);
}
```

## 14.7 Operators

Previously, the type system enforced that the operators are used with the right data types. For example, the addition was only allowed on numbers. Now, we need to define addition so that it works (to some extent) with any types. For instance, JavaScript has complicated coercion rules. If you add an empty array `[]` and a number `1` the result is a string `"1"`. For our compiler, we'll adopt simplified coercion rules. If the two operands are numbers, the result is a number, otherwise—`undefined`.

Using addition as our example, we can implement such operators like this:

```
visitAdd(node: Add) {
  node.left.visit(this);
  emit(`  push {r0, ip}`);
  node.right.visit(this);
  emit(`  pop {r1, ip}`);

  // Are both small integers?
  emit(`  orr r2, r0, r1`);
  emit(`  and r2, r2, #${tagBitMask}`);
  emit(`  cmp r2, #0`);

  emit(`  addeq r0, r1, r0`);
  emit(`  movne r0, #${undefinedBitPattern}`);
}
```

To implement the negation operator, we need to handle truthiness and falseness for the first time. With our tagging system, a word is falsy if it is zero, or ends with a falsy tag `0b10`. That's why we need to perform two comparisons.

```
visitNot(node: Not) {
  node.term.visit(this);

  // Is falsy?
  emit(`  cmp r0, #0`);
  emit(`  andne r0, r0, #${tagBitMask}`)
  emit(`  cmpne r0, #${falsyTag}`);

  emit(`  moveq r0, #${trueBitPattern}`);
  emit(`  movne r0, #${falseBitPattern}`);
```

```
}
```

We will need to do the falsy check in more places, so let's extract it into a helper method:

```
emitCompareFalsy() {
  emit(`  cmp r0, #0`);
  emit(`  andne r0, r0, #${tagBitMask}`)
  emit(`  cmpne r0, #${falsyTag}`);
}
```

Conditionals in `If` and `While` nodes will need to use `emitCompareFalsy` in palce of `cmp r0, #0` too.

## 14.8 Arrays

Array literals are mostly as before, with two exceptions. First, the length is now stored as a small integer. Second, a pointer tag is added as follows. Previously, `mov r0, r4` moved our call-preserved pointer to the return register `r0`. Now, we accomplish the same and add a `0b01` tag at the same time using `add r0, r4, #1`. We can do this because `malloc` will only return us word-aligned pointers with last two bits being `0b00`.

```
visitArrayLiteral(node: ArrayLiteral) {
  emit(`  ldr r0, =${4 * (node.args.length + 1)}`);
  emit(`  bl malloc`);
  emit(`  push {r4, ip}`);
  emit(`  mov r4, r0`);
  emit(`  ldr r0, =${toSmallInteger(node.args.length)}`);
  emit(`  str r0, [r4]`);
  node.args.forEach((arg, i) => {
    arg.visit(this);
    emit(`  str r0, [r4, #${4 * (i + 1)}]`);
  });
  emit(`  add r0, r4, #1`);  // Move to r0 and add tag
  emit(`  pop {r4, ip}`);
}
```

Array length primitive is almost the same as before, except that we load with a −1 offset to cancel the tag out.

```
visitLength(node: Length) {
  node.array.visit(this);
```

```
    emit(`  ldr r0, [r0, #-1]`);
  }
```

For array lookup, we don't use any tricks to remove the overhead of tagging and simply issue an additional instruction to remove the tag before the lookup. Also, we return `undefined` instead of zero if bound checking fails.

```
  visitArrayLookup(node: ArrayLookup) {
    node.array.visit(this);
    emit(`  bic r0, r0, #${pointerTag}`); // Remove tag
    emit(`  push {r0, ip}`);
    node.index.visit(this);
    emit(`  pop {r1, ip}`);
    // r0 => index, r1 => array, r2 => array length
    emit(`  ldr r2, [r1], #4`);
    emit(`  cmp r0, r2`);
    emit(`  movhs r0, #${undefinedBitPattern}`);
    emit(`  ldrlo r0, [r1, r0]`);
  }
```

Notice an interesting coincidence. Since we now store array length as a small integer, we don't need to shift left to convert array length to byte offset. Small integers are already shifted left by two thanks for the `0b00` tag.

---

Code generation for the `Function` node needs a small adjustment: it needs to return `undefined` instead of zero when a function ends with no return statement. The rest of the nodes (`Call`, `Return`, `Id`, `Assign`, `Var`, `Block`) do not require changes in code generation to adapt to dynamic typing.

With type checking turned on, our existing test suite should succeed unmodified. The only observable change is that integers are now 30-bit instead of 32-bit. However, if we turn the type checking off, we can express many more interesting programs, like the following one:

```
  isBoolean(x) {
    if (x == true) {
      return true;
    } else if (x == false) {
      return true;
```

```
  } else {
    return false;
  }
}

main() {
  var a = [];
  assert(a[1] + 2 == undefined);

  assert(!isBoolean(undefined));
}
```

# Chapter 15

# Garbage Collection

So far, we have used `malloc` to allocate memory dynamically, but we have never released it back to the operating system. You would need to call `free` to achieve that, otherwise your program memory consumption will quickly grow.

When a programmer needs to deallocate memory manually, it's called *manual memory management*. And for some languages that might be appropriate. However, most languages offer *automatic memory management* or, in other words, *garbage collection*.

The easiest way to incorporate a garbage collector into your compiler is to use an off-the-shelf component, such as Boehm–Demers–Weiser garbage collector (or Boehm GC) which is available as a library. You can link to this library and replace all the calls to `malloc` with `GC_malloc` from Boehm GC, and you're done!

But you are here not for the easy answers, are you? Right! Then, let's implement a simple garbage collector from scratch.

## 15.1 Cheney's algorithm

We will be implementing a garbage collection algorithm called Cheney's algorithm, named after Chris J. Cheney. Here's a simplified description of it. We will get to more details in the following sections.

At program startup, a large memory area is allocated (for example, using `malloc` from `libc`). This area is referred to as a *semi-space*. You will see why shortly. Our existing use of `malloc` for allocating arrays is replaced with a new function `allocate` which places the allocations (also called *blocks*) one after another in the semi-space. Once there's not enough memory in the semi-space for allocation, a function `collect` is run. It allocates a new semi-space and copies there only *live* objects from the old semi-space. The old semi-space is referred to as the *from-space*, and the new one is referred to as the *to-space*. The objects are *live* if they are reachable (by pointers) from the so-called *roots*. The *roots* are pointers that point to objects that we know we want to keep ahead of time. Examples of such pointers are pointers from the stack and global variables. When all live objects are copied to the to-space, the from-space is discarded (for example, using `free`).

When we run out of space again, we `malloc` a new semi-space. Now the old semi-space that used to be the to-space becomes the from-space, and the process repeats.

The first challenge is to figure out which objects are live. The other challenge is that when we copy blocks their addresses changes, and that means that we need to update all pointers that point to those objects.

## 15.2 Allocator

We will start by writing a custom allocator function that will replace our current use of `malloc`. Instead of calling `malloc` for each allocation, we will allocate a larger memory region and then will allocate many arrays (or other objects) there. Such a memory region is usually, confusingly, called a *heap*, but in the context of Cheney's algorithm, it is typically referred to as *semi-space*.

When talking about garbage collectors, we call individual spans of memory *blocks*. Our flat non-resizable arrays are precisely one block. Still, more complicated data structures (such as resizable arrays that we also talked about) could be implemented as several blocks connected with pointers. So, in this chapter, we'll use the term *block*, even though for our simple language it is synonymous with an array.

So, at program startup, we allocate a large space using `malloc`. We

need to manage three pointers:

- The pointer returned by `malloc` that we will call *semi-space start*.
- The *allocation pointer* which points to the first word in a space that is not occupied by a block.
- The pointer to the end of the semi-space (which is *semi-space start* plus the size of the space) that we'll call *semi-space end*.

We can store these pointers in a `.data` section of our program, or we can store all (or some) of them in call-preserved registers. (We haven't talked about using call-preserved registers this way, but they can be used to hold global-like variables if we don't save them on the stack.)

The start and end pointers will not change until we create a new space. The allocation pointer, however, will change with each allocation. It starts out pointing at the same address as the semi-space start, and with each allocation moves towards the semi-space end.

Now we can implement the `allocate` function. When it is called (with a number of bytes to allocate), we check if the area between the allocation pointer and the space end is enough to hold it. Then, we increment the allocation pointer by the requested number of bytes and return the previous value of the allocation pointer as the result. In other words, with each allocation, we pack blocks one after another from semi-space start to semi-space end.

After several allocations, the state of our program may look like the following figure.

We have a stack with three frames allocated. The frames are connected with frame pointers. These frames have several pointers to the blocks that we have just allocated in our semi-space. Those blocks have pointers between them (for example, an array has a nested array). The pointers may have cycles (an array element points to the array itself). The non-pointer words (left blank) hold some scalar data like numbers or booleans (both on the stack and in the semi-space).

We also make sure that the blocks have their length as the first word (as we did with arrays). We call such word in a block a *header*. (Headers usually occupy one or two words and can contain more data, such as object type, for example.) Instead of writing the length of the block in the header, we denote the header with `H` and

group the words in the same block using rounded corners. This way, we can also see the block length in the diagram. As you can see, we have allocated five blocks of lengths two and three words.

Let's say that the program wants to allocate a block of four words now. There's not enough space in the semi-space (from the allocation pointer to the end of the semi-space). This condition triggers the process of garbage collection.

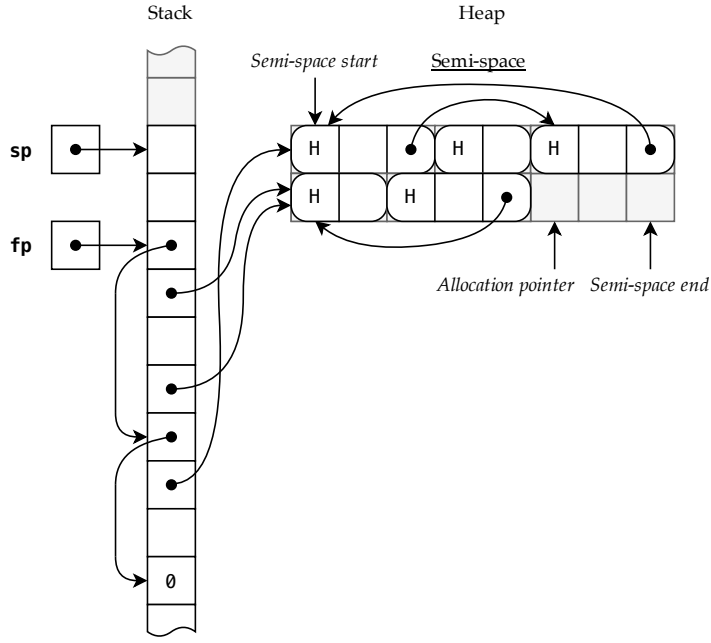


Figure 15.1: The state of our program: stack and heap

## 15.3 Collection

We start by allocating a new semi-space called *to-space*, while we refer to the old one as the *from-space*. They are named that because blocks are copied *from* the from-space *to* the to-space. In the next figure, you can see the two semi-spaces. They both use three pointers: *start*, *allocation*, and *end* pointers.

We will walk down the stack, frame-by-frame, and look at each word in a frame: is it a pointer or not? With our existing tagging scheme, we can easily distinguish pointers from scalars. This process is called *scanning*, and we use the *scan pointer* to track where we are right now in the scanning process.

You can see the *scan pointer* and the newly allocated *to-space* on the next figure.

> **Note:**
>
> If we didn't use tagging, we could check if the word's numeric value is between the from-space start and end. If it's within that range, we can conclude that the value is a pointer if it's not we treat it as a scalar. This is not always precise: our program might deal with some numeric value that happens to be in that range. In this case, the algorithm might keep a block (or several) in memory which is garbage. But this is not a problem, in practice. Such garbage collectors are called *conservative*. Boehm GC is an example of such a collector.

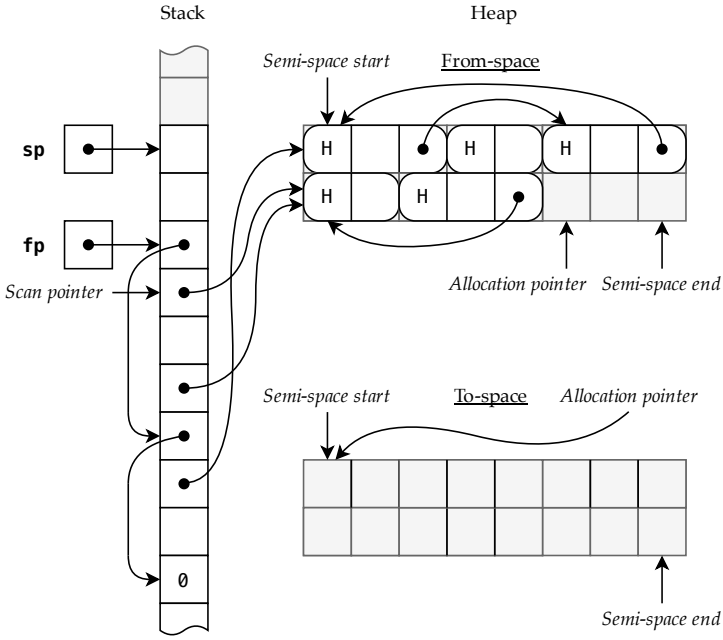


Figure 15.2: The two semi-spaces and their pointers

As we scan down the stack, we encounter our first pointer. We copy the corresponding block to the to-space and bump the value of the to-space allocation pointer. In the next figure, you can see the two-word block copied to the to-space. The pointer on the stack is updated to point to the new block. (The previous value of the pointer is shown with a dashed line.) We also update the same block in the from-space and replace the header word with a so-called *forwarding pointer*, shown in grey.

If in future we encounter other pointers pointing to the block that we just moved, we will need to update them with a new pointer. That's why we store the forwarding pointer in the old block that points to the new block. It's like when you move to a new address and leave a memo at the old one that you have moved, so your mail can be forwarded. When we reach a block, we can also distinguish whether it starts with the header (number) or a forwarding pointer (pointer) using our tagging scheme. Of course, the block length is lost, but we won't need it.

Since we allocated a block in the to-space, the allocation pointer is incremented by the size of the block.

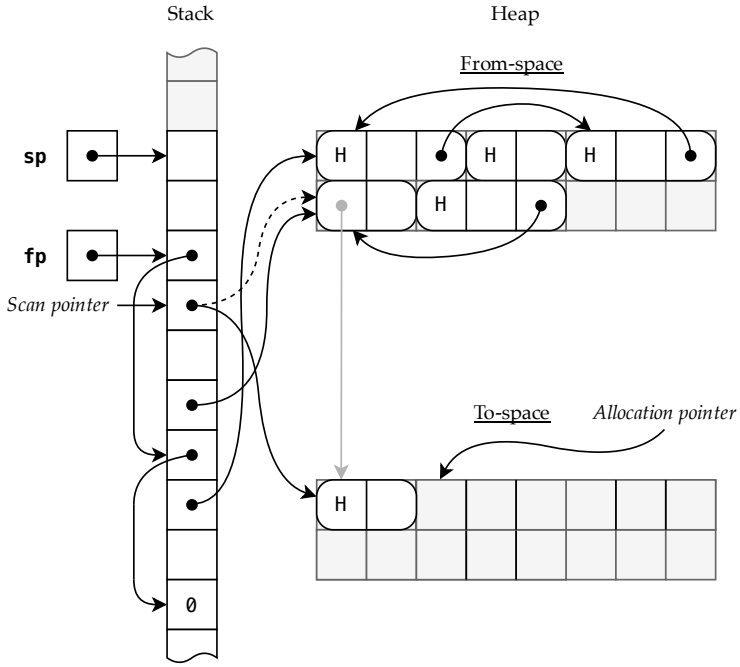


Figure 15.3: Scanninng the stack: first pointer

In the same frame, we skip over one word which holds a scalar, and then we encounter another pointer. By following that pointer, we discover that it starts with a forwarding pointer instead of a header. You can recognize that it's the same block that we have just moved. This time, we do not copy the block. Instead, we update the value on the stack with the value of the forwarding pointer.

---

Stacks also hold values of the saved link registers. We know their location is next to the saved frame pointers and can ignore them. Or, in our case, we don't need special handling for them: since link registers are not tagged, they will be interpreted as scalars and skipped anyway.

Another question is—how far down the stack should we go? At some point, we'll need to stop. At the start of the program, in `main`, we can allocate a zero byte on the stack and point the frame pointer there. This way, we can find out which is the last frame in our program and stop. There are also ways to achieve that without frame pointers.

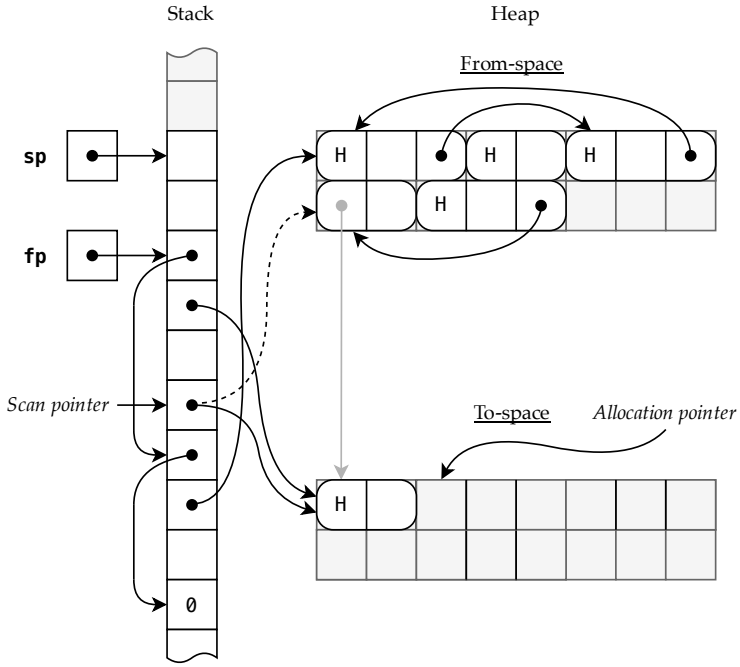


Figure 15.4: Scanninng the stack: second pointer

We move to the next (and last) frame and start scanning again. We encounter our last pointer on the stack. Same as before, we copy the pointed block from the from-space to the to-space and replace the old block's header with a forwarding pointer. The forwarding pointer is in grey, and the old value of the pointer is shown with a dashed arrow.

We have scanned the whole stack, and we should do similarly with other roots, such as global variables and registers. To scan registers, we can save them on the stack before scanning the stack. However, we are not finished: there's a block in the to-space that points to another block in the from-space.

Similarly to how we scanned the stack, we now start scanning the to-space by looking at one word at-a-time and checking if it's a pointer. If it's a pointer to a header—we copy the block to the to-space. If it's a pointer to a forwarding pointer, we update the pointer to that value.

> **Note:**
>
> Sometimes headers hold additional data for the garbage collector. For example, a tag in the header can mean that the whole block consists only of scalars and should be skipped by the collector to save time. This is very relevant, for example, for handling large strings.
>
> Also, when pointer tagging is not used, each block (and frame) can hold a field with information about which word is a pointer or not. This could be done in the form of a bit map, with one bit per word. In other words, the tagging bits can be moved from each word to a central location in a block.

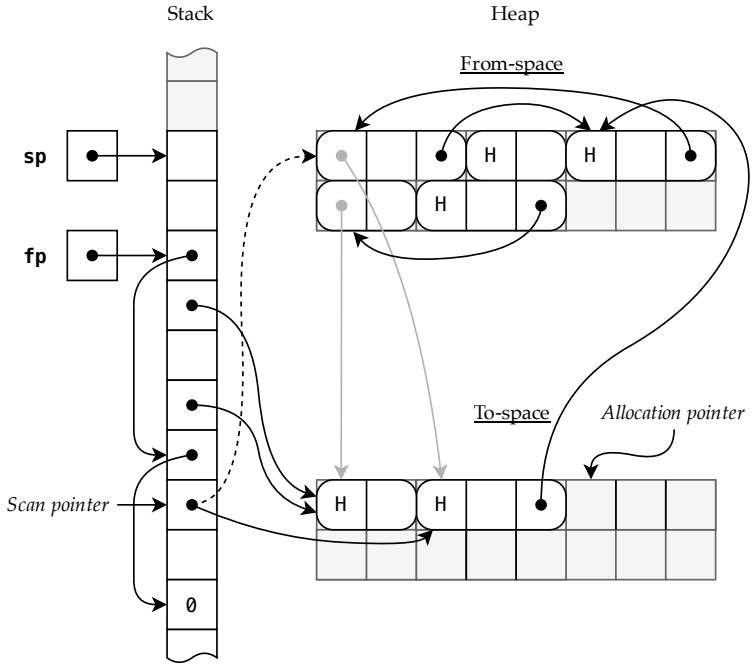


Figure 15.5: Scanninng the stack: third pointer

Now, we point the scan pointer to start of the to-space and start scanning one word at-a-time.

As we scan, we encounter our first pointer. We allocate the pointed block on the to-space, update the pointer to point to the newly allocated block, and store a forwarding pointer (grey) in the old block.

You can see that, as we continue scanning the to-space, it grows, so now we have more scanning to do.

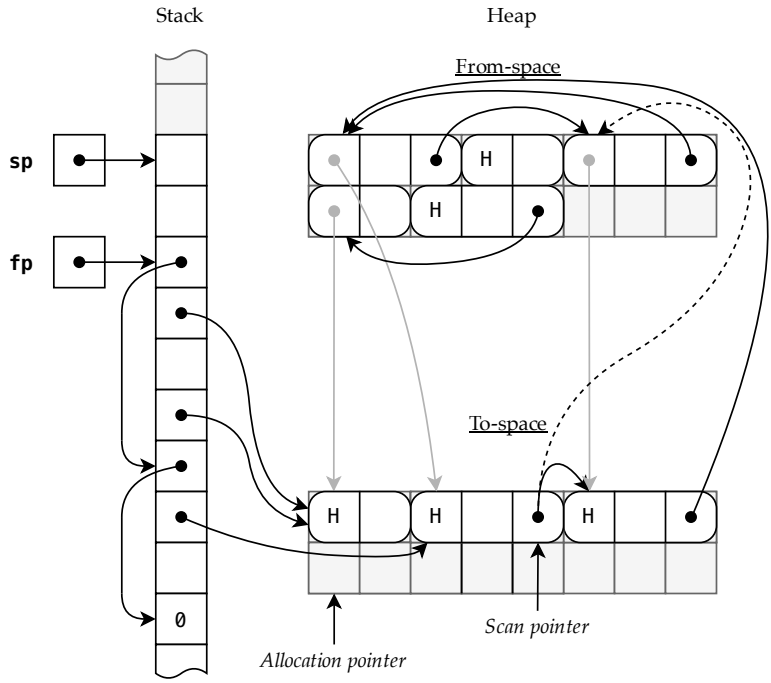


Figure 15.6: Scanninng the to-space: first pointer

As we scan the newly-copied block, we encounter another pointer. It points to a block in the from-space that has a forwarding pointer, so we only update the pointer to the new value, without copying (since that block has already been copied). Once we finish that, there are no more blocks to scan—we have completed the collection phase. We can see that there are two blocks in the from-space (now highlighted grey) that were not copied. That means they are garbage. Now we can dispose of the from-space (using `free`) and, finally, allocate that four-word block that we wanted to allocate in the very beginning before the collection started. The program may now resume.

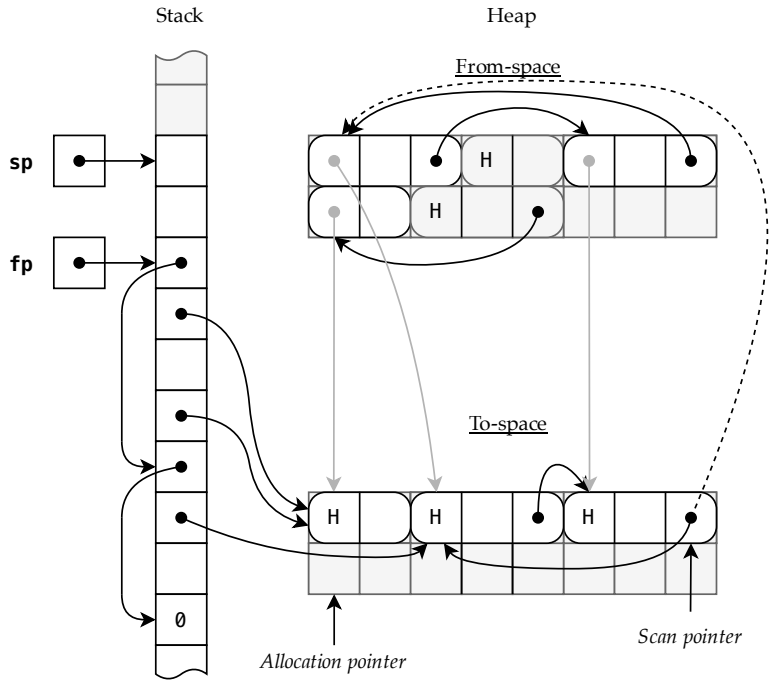


Figure 15.7: Scanninng the to-space: second pointer

In the next figure, you can see the state of the program after the collection has finished, and the from-space deallocated.

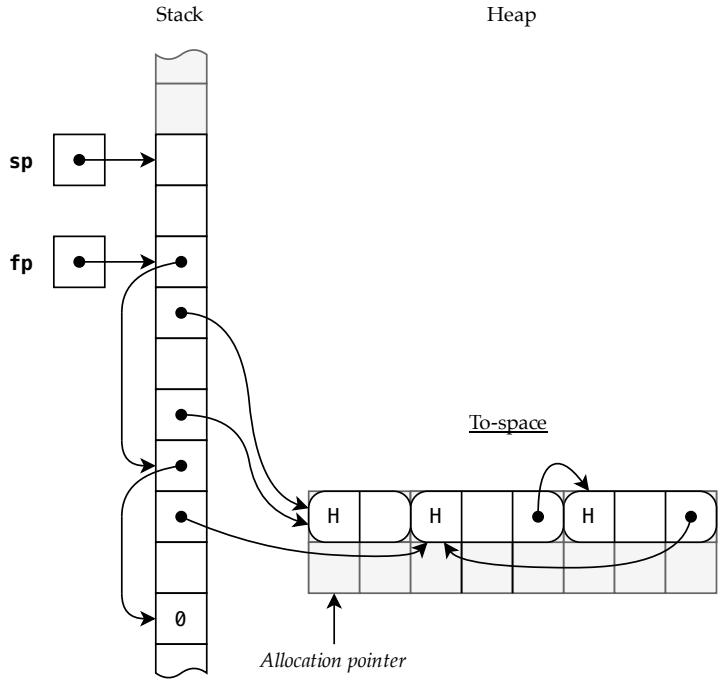


Figure 15.8: The state of our program after collection

## 15.4 Discussion

In this example, our semi-spaces were only 16 words or 64 bytes. In reality, we need to allocate a larger area, and the larger it is, the fewer times we will need to collect garbage.

Now that we had an in-depth view of Cheney's algorithm let's discuss some of the properties of such garbage collector.

First, it offers fast allocation. Most of the time, allocation only needs to check that there's enough space left, and increment the allocation pointer—that's it! This can be implemented in a handfull of instructions and can be made even faster if the allocation pointer is assigned a dedicated call-preserved register. Compare this to `malloc` where typical implementation needs to execute 50 to 200 instructions before returning.

Second, it's a *compacting* or *moving* collector. After the collection phase, all the blocks are allocated one-after-another without holes or fragmentation. This is good for data locality and cache performance.

But not everything is perfect with this algorithm.

It's a *stop-the-world*, *non-concurrent* collector. When collection happens, the program is paused until the collection finishes. In a concurrent collector, the collection happens a bit-at-a-time with each allocation, to avoid long pauses.

Another property of this algorithm is that during collection, when both semi-spaces are allocated the amount of memory consumed is twice as high.

Our collector is also *non-generational*.

## 15.5 Generational garbage collection

It is observed that new blocks often become garbage fast. And then some blocks stay around for a long time. We want to scan new blocks more frequently to reclaim memory and to scan older blocks more rarely (since they are less likely to be garbage). This can be achieved with *generational* collectors.

For example, a collector with two generations G0 and G1 may have two heaps or spaces: a smaller one for G0 and a larger one for

G1. Collections happen more frequently in G0 and more rarely in G1. They can also use different algorithms. All new objects are allocated in G0, and after G0 collection happens, the blocks that remained are copied to G1.

Cheney's algorithm (and variations) is often used for G0 collectors because of fast allocation and because it is a moving collector. In such generational collector, we would always copy blocks into G1 heap, and not into a semi-space.

# Chapter 16

# Appendix A: Running ARM Programs

## 16.1 32-bit Linux on ARM (e.g. Raspberry Pi)

If you're a lucky owner of an ARM board like a Raspberry Pi, you don't need any special steps. You just use the built-in `gcc` toolchain and run the produced executables natively, as discussed in the book.

For example, assuming you have an assembly listing `hello.s` from *Part I*, you can assemble it like this:

```
$ gcc hello.s -o hello
```

And then you can run the produced executable natively:

```
$ ./hello
Hello, assembly!
```

## 16.2 64-bit Linux on ARM64

If you are running 64-bit Linux on an ARM64 development board or an ARM64 server or compute instance, then the good news is that ARM64 is backwards-compatible and can run 32-bit ARM ex-

ecutables natively. (Note, however, that there are instances when this backwards-compatibility is disabled by the board/server manufacturer to save costs.)

However, the built-in `gcc` toolchain will produce ARM64 executables and will require ARM64 assembly. That means you need to install a cross-compilation version of a `gcc` toolchain. Assuming a Debian-based Linux distro with `apt-get` package manager, you can install it as follows:

```
$ apt-get install gcc-arm-linux-gnueabihf
```

You might need to use `sudo` to install it.

Now, whenever the book uses `gcc`, you need to use `arm-linux-gnueabihf-gcc -static` instead. For example, assuming you have an assembly listing `hello.s` from *Part I*, you can assemble it like this:

```
$ arm-linux-gnueabihf-gcc -static hello.s -o hello
```

And then you can run the produced executable natively:

```
$ ./hello
Hello, assembly!
```

## 16.3 Linux on x86-64 using QEMU

This assumes that you don't have access to an ARM Linux machine like a Raspberry Pi or an ARM server, but only to a good old x86-64 machine based on an AMD or Intel processor.

The following has been tested on Ubuntu 20.04 LTS, but it should work the same on all Debian-based Linux distros. For other distros, you might need to use a different package manager. The package names could be slightly different, as well.

You need to install two packages. First, `gcc-arm-linux-gnueabihf` is a version of `gcc` toolchain that cross-compiles to ARM. In other words, even though you are running an `x86-64` processor, it will produce ARM binaries (which you wouldn't be able to run directly on `x86-64`).

```
$ apt-get install gcc-arm-linux-gnueabihf
```

Second, you need to install QEMU–a piece of software that allows emulating different processors, including ARM. However,

we won't be using the `qemu` package, but instead, one called `qemu-user`. While `qemu` allows emulating complete machines, `qemu-user` is made to emulate individual executables.

```
$ apt-get install qemu-user
```

We could have used the `qemu` package, created some hardware configuration, logged into the machine, and so on. On the other hand, `qemu-user` allows us to run ARM executables as if we were on an ARM machine.

By the way, depending on your setup, you might need to use `sudo` to install these packages.

Now, whenever the book uses `gcc`, you need to use `arm-linux-gnueabihf-gcc -static` instead. For example, assuming you have an assembly listing `hello.s` from *Part I*, you can assemble it like this:

```
$ arm-linux-gnueabihf-gcc -static hello.s -o hello
```

Then, the next step is no mistake:

```
$ ./hello
Hello, assembly!
```

What just happened? How are we running ARM assembly on x86-64? It turns out `qemu-user` has a smart mechanism: when installed, it configures itself to handle ARM binary files. However, on some Linux configurations, this won't work, and you need to be a little bit more explicit about invoking QEMU:

```
$ qemu-arm ./hello
Hello, assembly!
```

Note that even though the package name is `qemu-user`, the executable name that we're interested in is `qemu-arm`, since ARM is only one possible target of QEMU.

## 16.4 Windows on x86-64 using WSL and QEMU

Follow the steps to enable Windows Subsystem for Linux (WSL). Ubuntu 20.04 LTS has been tested for this purpose.

*https://ubuntu.com/wsl*

Now, open a WSL terminal and follow the steps described in the previous section, *Linux on x86-64 using QEMU*.

# Chapter 17

# Appendix B: GAS *v.* ARMASM Syntax

Two different syntaxes are used for ARM assembly. The first one is the GNU Assembler (GAS) syntax. It is also supported by the official ARM toolchain based on Clang called `armclang`.

The other one is the legacy ARMASM syntax. Most likely, you will not need to deal with it, but I include a small Rosetta Stone–style (side-by-side) comparison cheat sheet for completeness.

## 17.1 GNU Assembler Syntax

```
/* Hello-world program.
   Prints "Hello, assembly!" and exits with code 42. */

.data
  hello:
    .string "Hello, assembly!"

.text
  .global main
  main:
    push {ip, lr}

    ldr r0, =hello
    bl printf

    mov r0, #41
    add r0, r0, #1  // Increment 41 by one.

    pop {ip, lr}
    bx lr
```

## 17.2 Legacy ARMASM Syntax

```
; Hello-world program.
; Prints "Hello, assembly!" and exits with code 42.

  AREA ||.data||, DATA
hello
  DCB  "Hello, assembly!",0

  AREA ||.text||, CODE
main PROC
  push {ip, lr}

  ldr r0, =hello
  bl printf

  mov r0, #41
  add r0, r0, #1  ; Increment 41 by one.

  pop {ip, lr}
  bx lr
```