An Introduction to Imperative Programming

An Introduction to Imperative Programming

Sebastian Hack
Saarland University

August 4, 2024

[1] `creativecommons.org/licenses/by-sa/4.0`

Preface

This material serves as lecture notes and background reading for the second-semester course “Programming 2” that I regularly teach since 2008 in the Computer Science Bachelor’s programs at Saarland University[2].

Our curriculum emphasizes a solid programming education that does not only cover practical aspects but also an introduction to foundations such as structure and semantics of programming languages, type systems, program correctness, and verification. As the name “Programming 2” suggests, this course follows “Programming 1”, an introduction to functional programming and computer science in general and it precedes an introduction to algorithms and data structures.

Programming 2 (and therefore this text) has the following objectives:

- *Provide solid practical skills in statically-typed practically-relevant imperative and object-oriented programming languages.*

  To this end, the course is accompanied by a series of programming projects in all languages covered (MIPS, C, Java) that increase in size and difficulty.

- *Provide an introduction to simple algorithms and data structures from an imperative perspective.*

  One part of learning the practice of programming is also to learn about data structures and algorithms that solve interesting problems. We try to provide this experience through non-trivial programming projects that often rely on clever data structures and algorithms that make the project feasible in the first place (shortest paths in route planning, acceleration structures in ray tracing, dynamic programming, etc.).

- *Provide an overview of and experience with the software stack from machine code to a high-level language.*

  I consider it important that students are able to identify the different levels of abstractions that are at work in programming and they can mentally move between these abstractions without problems. Seeing a C/C++ program and actually having a vague idea of what happens on the machine is not only helpful to eliminate “cargo cult” thinking but also vital when it comes to performance or security considerations. Therefore, the course intentionally starts with computer arithmetic and machine code programming to convey an impression on how programs execute at the lowest level. This enables an understanding of the abstractions higher-level imperative languages provide to make programming machines more bearable and scalable, what these abstractions actually abstract from, and what the cost of these abstractions are.

[2]`uni-saarland.de`

- *Provide an introduction to the theoretical foundations of imperative programming languages and a basic understanding of program correctness, testing, and verification.*

  Being able to understand how computers execute is one thing. It is equally important to understand how to describe programs and their execution in mathematical terms to be able to talk about concepts like termination, undefined behavior, correctness, etc. in a meaningful way. It is important to understand that the details of these notions are so intricate, subtle, and important that it is impossible to leave it at hand waving. Additionally, one can make the experience that the process of formalizing things uncovers and clarifies ambiguities and makes precise what one is actually talking about.

Appendix C presents and discusses a sample course structure using this text. In this sample structure, some of the sections of this book are discussed in greater detail than others.

Happy hacking!

Acknowledgements

I am indebted to many people who helped to develop this course and the accompanying material. In alphabetical order I would like to thank Johannes Doerfert, Timo Gros, Tina Jung, Christoph Mallon, Fabian Ritter, Julian Rosemann, Sigurd Schneider, Marcel Ullrich for their hard work over many years in preparing exercises, finding and fixing bugs, reviewing and critically discussing the contents of this course.

I would like to thank Jörg Hoffmann for preparing a complete slide deck for these lecture notes, for his many many helpful comments, and his relentless scrutiny.

Contents

Chapter 1

Computer Arithmetic

Most computers are digital (from Latin *digitus* = finger). This means that a piece of information is represented by a symbol, called **digit**, drawn from a finite set. (There are also analog computers where information is represented by a continuous quantity like voltage.) Almost all digital computers are *binary* which means that the set of symbols has two elements commonly called 0 and 1. A binary digit is called **bit**.

Aside: Making computers binary is basically motivated by the way they are implemented in hardware: A 1 is represented by a certain voltage level and 0 is represented by another one (typically 0 volts).

One important property of computers is that hardware typically operates on bit strings of *fixed* size and is therefore not able to work with arbitrarily large numbers *directly*. This size is called the **word size** of the machine. Many programming languages reflect this fact in their type systems by providing types like `uint32_t` which stands for a natural number between 0 and $2^{32} - 1$ in the C programming language.

Computers define a special kind of arithmetic of these finite-length bit strings called **two's-complement arithmetic**. Two's-complement arithmetic is pre-dominant in modern computers and at the core of the integer types of many modern programming languages. It allows to interpret bit strings as **unsigned** (natural) numbers or **signed** (integers) numbers and defines the appropriate operations for both interpretations.

Understanding the details of two's-complement arithmetic is important to understand the effects of the computations in computer programs, especially in corner cases when overflows happen or in low-level settings when debugging code or analyzing machine code programs.

1.1 Positional Notation

It is common for us to write numbers as finite sequences of digits. Here, we don't care about the actual symbol (like $1, 2, \ldots$) we use for a digit but only that we have finitely many of them. So for now, we assume that we have B many digits and, for the sake of simplicity, we identify them with the natural numbers $0, \ldots, B - 1$ where B is called the **base** of a specific positional number system. In our daily lives we use base 10 systems, a binary computer uses a base 2 system.

Formally, a sequence of digits x of length n is an n-tuple of digits, i.e. an element of the set $\{0, \ldots, B-1\}^n$. To simplify the presentation, we write an n-tuple $(x_{n-1}, \ldots, x_0)$ shortly as $x_{n-1} \ldots x_0$.

Up to now a sequence of digits is just a bunch of symbols and does not have a particular meaning. The following definition defines a function, called the **unsigned interpretation** that associates a natural number to each finite digit sequence.

Definition 1.1.1 Unsigned Interpretation of Digit Sequences.

$$\langle\cdot\rangle_B : \{0,\dots,B-1\}^+ \longrightarrow \mathbb{N} \qquad x_{n-1}\dots x_0 \mapsto \sum_{i=0}^{n-1} x_i B^i$$

◊

x_{n-1} is called the **most significant digit** and x_0 **the least significant digit**.

Given two digit sequences $\alpha = a_{n-1}\dots a_0$ and $\beta = b_{m-1}\dots b_0$, we can concatenate them and denote this by juxtaposing them $\alpha\beta = a_{n-1}\dots a_0 b_{m-1}\dots b_0$. We abbreviate the digit sequence $\underbrace{a\dots a}_{n}$ by a^n.

Lemma 1.1.2 *Let $\alpha = a_{n-1}\dots a_0$ and $\beta = b_{m-1}\dots b_0$ be two digit sequences. Then $\langle\alpha\beta\rangle_B = \langle\alpha\rangle_B \cdot B^m + \langle\beta\rangle_B$*

The definition of $\langle\cdot\rangle_B$ matches our intuition and coincides with the way we use numbers in our daily life. However, we might be interested in demonstrating that indeed each natural can be represented by a digit sequence. This is established by the next lemma.

Lemma 1.1.3 *Let B be greater than one and n greater than zero. For each natural number $k \in \{0,\dots,B^n-1\}$ there is a digit sequence α of length n such that $\langle\alpha\rangle_B = k$*

Proof. By induction over n.

1. *Base case $n = 1$.*

   Let $k \in \{0,\dots,B-1\}$. Then, k is a digit itself and $\langle k\rangle_B = k$.

2. *Induction step $n \longrightarrow n+1$.*

   Consider a number $k \in \{0,\dots,B^{n+1}-1\}$. Let q be the maximal number such that there exists an r with $k = q \cdot B^n + r$. Because q is maximal $q \in \{0,\dots,B-1\}$ and $r \in \{0,\dots,B^n-1\}$.

   By induction, there is a digit sequence ρ such that $\langle\rho\rangle_B = r$. By Lemma 1.1.2 it holds that $\langle q\rho\rangle_B = k$.

■

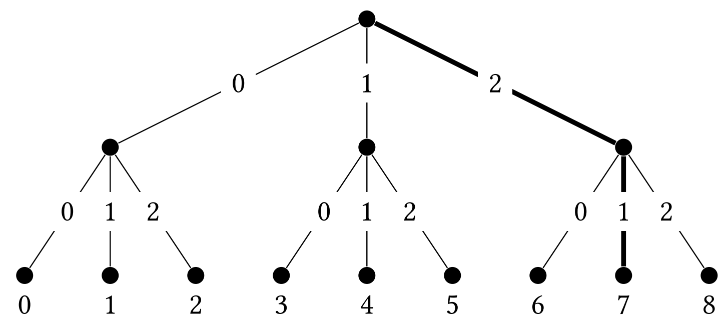


Figure 1.1.4 This figure gives some intuition for Lemma 1.1.3. A digit sequence of a base B can be seen as a path in a tree in which each inner node has B children. The path shown in bold is the number $\langle 21\rangle_3 = 7$. Each layer of tree edges corresponds to a position, less significant digits being further down. One can also see that using positional notation is very efficient because to represent a number n one only needs $\lceil\log_B n\rceil$ many digits.

Checkpoint 1.1.5 Base 1. What is the difference between a $B = 1$ positional system and a $B > 1$ positional system?

Lemma 1.1.6 $2^n = 1 + \sum_0^{n-1} 2^i$

Proof. By induction over n.

1. *Base case* $n = 1$.

   Trivial.

2. *Induction step* $n \longrightarrow n + 1$.

   The induction hypothesis is $2^n = 1 + \sum_0^{n-1} 2^i$. Adding 2^n to each side gives the desired result.

■

1.2 Binary Numbers

In the following, we will limit ourselves to binary numbers and therefore set $B = 2$. We will also write $\langle x \rangle$ instead of $\langle x \rangle_2$. We also have only two digits $\mathbb{B} := \{0, 1\}$. An element of $\mathbb{B}$ is called **bit**. An element of $\mathbb{B}^8$ is called **byte**. The **complement** $\overline{b}$ of a bit b is defined as $\overline{b} := 1 - b$. A sequence of bits is called a **bit string**. In a bit string $b_{n-1} \dots b_0$, bit b_{n-1} is called the **most significant bit** and b_0 is called the **least significant bit**.

We define three binary operations on bits **and**, **or**, and **xor** (exclusive or) by means of a value table. These primitive operations are enough to formulate the addition of **bit strings** (sequences of bits).

		and	or	xor
a	b	a & b	$a \mid b$	a ˆ b
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Checkpoint 1.2.1 Elementary Operations on Bits. There are actually operations that are more elementary than and, or, xor. One of them is nand $\overline{a \,\&\, b}$ which is 0 if $a = b = 1$ and 1 otherwise. Show how to express and, or, xor using nand. What other elementary operations like nand are there?

Solution. This is the truth table for nand.

a	b	$\overline{a \,\&\, b}$
0	0	1
0	1	1
1	0	1
1	1	0

We can express negation using nand:

a	$\overline{a \,\&\, a}$	$\overline{a}$
0	1	1
1	0	0

We can express conjunction (and) using nand. We can use plain old negation, since we already know how to encode it just using nand.

a	b	$\overline{\overline{a \mathbin{\&} b}}$	$a \mathbin{\&} b$
0	0	0	0
0	1	0	0
1	0	0	0
1	1	1	1

We can now express disjunction (or) using nand. Again, we also use negation and conjunction since we know how to express them.

a	b	$\overline{a}$	$\overline{b}$	$\overline{\overline{a} \mathbin{\&} \overline{b}}$	$a \mid b$
0	0	1	1	0	0
0	1	1	0	1	1
1	0	0	1	1	1
1	1	0	0	1	1

We have already shown that we can express and and or with nand. So it is enough to show that xor can be expressed with and and or to prove that we can only express it with nand.

a	b	$\overline{a}$	$\overline{b}$	$\overline{a} \mathbin{\&} b$	$a \mathbin{\&} \overline{b}$	$(\overline{a} \mathbin{\&} b) \mid (a \mathbin{\&} \overline{b})$	$a \mathbin{\hat{}} b$
0	0	1	1	0	0	0	0
0	1	1	0	1	0	1	1
1	0	0	1	0	1	1	1
1	1	0	0	0	0	0	0

Another elementary operation like nand is nor. It is 1 if $a = b = 0$ and 0 otherwise.

Checkpoint 1.2.2 Nand is elementary. Show that nand is indeed universal, i.e. *every* boolean function $f(a, b)$ can be expressed only using nands.

Hint. First, show that every boolean function can be represented using and, not and or. Then use the other exercise to relate this to nand. You may also need to prove that not can be expressed using nand.

Solution. In total, there are 16 different possibilities for boolean functions with two arguments: Since $f(a, b)$ takes two arguments, each of which can be either 0 or 1, there are exactly 4 possible different inputs. For each of these inputs there are 2 different possible results: 0 and 1. This means in total we can find $2^4 = 16$ functions. All of functions can be represented only using and, or, and negation, as follows:

$f(0,0)$	0	1	0	1	0	1	0	1
$f(0,1)$	0	0	1	1	0	0	1	1
$f(1,0)$	0	0	0	0	1	1	1	1
$f(1,1)$	0	0	0	0	0	0	0	0
$f(a,b)$	$a \mathbin{\&} \overline{a}$	$\overline{a} \mathbin{\&} \overline{b}$	$\overline{a} \mathbin{\&} b$	$\overline{a}$	$a \mathbin{\&} \overline{b}$	$\overline{b}$	$(a \mathbin{\&} \overline{b}) \mid (b \mathbin{\&} \overline{a})$	$\overline{a} \mid \overline{b}$

$f(0,0)$	0	1	0	1	0	1	0	1
$f(0,1)$	0	0	1	1	0	0	1	1
$f(1,0)$	0	0	0	0	1	1	1	1
$f(1,1)$	1	1	1	1	1	1	1	1
$f(a,b)$	$a \mathbin{\&} b$	$(a \mathbin{\&} b) \mid (\overline{a} \mathbin{\&} \overline{b})$	b	$b \mid \overline{a}$	a	$a \mid \overline{b}$	$a \mid b$	$a \mid \overline{a}$

Now that we have shown that every boolean function with two arguments can be represented with and, not, and or and that we have shown in exercise 6 that and

and or can be expressed with nand, the only thing left to show is that not can also be expressed using only nand.

a	$\overline{a}$	$\overline{a \mathbin{\&} a}$
0	1	1
1	0	0

In conclusion, we have now shown that and, or and not can be expressed using only nand, because we have also shown that every boolean function with two arguments can be represented by and, or and not we can conclude that nand is universal.

Hexadecimal Numbers. It is common to write bit strings more compactly by grouping four bits together. There are 16 possible combinations of four bits, so four bits can encode 16 different numbers, and therefore, a packet of four bits can be represented by one symbol of the following set $\{0, \dots, 9, \mathrm{a}, \dots \mathrm{f}\}$. Defining

$$\langle a \rangle_{16} = 10, \dots, \langle f \rangle_{16} = 15$$

gives numbers in a base 16 system, so-called **hexadecimal numbers**.

Example 1.2.3

$$\begin{aligned}\langle \mathrm{ff} \rangle_{16} &= \langle 11111111 \rangle_2 = \langle 255 \rangle_{10} \\ \langle 80 \rangle_{16} &= \langle 10000000 \rangle_2 = \langle 128 \rangle_{10} \\ \langle 19\mathrm{fa} \rangle_{16} &= \langle 0001100111111010 \rangle_2\end{aligned}$$

□

Many programming languages support hexadecimal numbers. In C and languages that are syntactically inspired by C, you can use the prefix `0x` to denote a hexadecimal number such as `0x19fa`. Another common prefix used by some assembly lanaguges and Pascal-like languages is `$` as in `$19fa`.

1.3 Addition and Subtraction

Adding two binary numbers works in the same way as adding decimal numbers. Let us first consider the sum $a + b$ of two bits a and b. In general, the sum of two bits is two bits long, because the addition can cause a carry. Bit 0 of the sum is given by the xor of both bits: $a \mathbin{\hat{}} b$. The **carry** c of that position is 1 if and only if $a = b = 1$, i.e. $a \mathbin{\&} b = 1$.

When adding entire bit strings not just single digits, the carry at a position needs to be propagated into the next higher position. So, if c_i denotes the carry bit position $i - 1$ generates, the sum bit at position i is given as

$$s_i \coloneqq a_i \mathbin{\hat{}} b_i \mathbin{\hat{}} c_i \tag{1.1}$$

The carry bit of position $i - 1$ may also influence the carry bit c_{i+1} out of position i. For example, if $a_i = b_i = c_i = 1$, then $c_{i+1} = 1$. The carry bit position i generates is defined as

$$c_{i+1} \coloneqq (a_i \mathbin{\&} b_i) \mid (a_i \mathbin{\&} c_i) \mid (b_i \mathbin{\&} c_i) \tag{1.2}$$

The following lemma proves that $c_{i+1}s_i$ indeed is the sum of a_i, b_i, and c_i.

Lemma 1.3.1 $a_i + b_i + c_i = \langle c_{i+1} s_i \rangle$

Proof. By value table:

a_i	b_i	c_i	$a_i + b_i + c_i$	c_{i+1}	s_i	$\langle c_{i+1} s_i \rangle$
0	0	0	0	0	0	0
1	0	0	1	0	1	1
0	1	0	1	0	1	1
1	1	0	2	1	0	2
0	0	1	1	0	1	1
1	0	1	2	1	0	2
0	1	1	2	1	0	2
1	1	1	3	1	1	3

■

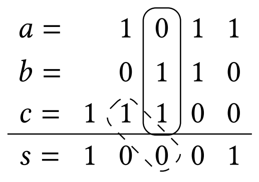


Figure 1.3.2 The bits in the dashed box represent the right-hand side of Lemma 1.3.1 and the digits in the solid box the ones on the left-hand side.

Now, consider two bit strings a and b of equal length n. When adding a and b we assume that there flows no carry into the least-significant digit, i.e. $c_0 = 0$.

Definition 1.3.3 Addition of bit strings. $a + b \coloneqq c_n s_{n-1} \dots s_0$ ◊

Remark 1.3.4 Efficiency and practicability of our adder. While Lemma 1.3.5 shows that the addition algorithm we defined is sound, it has a significant practical disadvantage. Our algorithm is a so-called **ripple-carry adder** which means that the carry "ripples" through the positions: In order to compute s_i you have to have computed c_i for which you need s_{i-1} and so forth. This makes the depth of the actual circuit of ands, ors, xors that implements the adder linear in terms of the number of positions. Since the circuit depth significantly influences the frequency by which you can clock the computer and therefore its speed, practical implementations use different adders, so called **carry-lookahead adders** that have more gates but are flatter and can therefore be clocked faster.

The following lemma shows that this definition is sound, i.e. the bit operations by which we defined the addition really produce a bit string that represents the natural number of the sum of the natural numbers that are represented by the individual bit strings.

Lemma 1.3.5 Addition of bit strings is sound. *Let a and b be two bit strings of length n. Then, $\langle a + b \rangle = \langle a \rangle + \langle b \rangle + c_0$.*

Proof. By induction over n.

1. *Base case $n = 1$.*

   Follows directly from Lemma 1.3.1.

2. *Induction step $n - 1 \longrightarrow n$.*

   The induction hypothesis is

   $$\langle c_{n-1} s_{n-2} \dots s_0 \rangle = \langle a_{n-2} \dots a_0 \rangle + \langle b_{n-2} \dots b_0 \rangle$$

   By using the definitions and lemmas we have established above, we get:

   $$\langle a \rangle + \langle b \rangle = a_{n-1} 2^{n-1} + \langle a_{n-2} \dots a_0 \rangle + b_{n-1} 2^{n-1} + \langle b_{n-2} \dots b_0 \rangle \quad \text{Lemma 1.1.2}$$

$$
\begin{aligned}
&= (a_{n-1} + b_{n-1}) \cdot 2^{n-1} + \langle a_{n-2} \dots a_0 \rangle + \langle b_{n-2} \dots b_0 \rangle && \\
&= (a_{n-1} + b_{n-1}) \cdot 2^{n-1} + \langle c_{n-1} s_{n-2} \dots s_0 \rangle && \text{IH} \\
&= (a_{n-1} + b_{n-1} + c_{n-1}) \cdot 2^{n-1} + \langle s_{n-2} \dots s_0 \rangle && \text{Lemma 1.1.2} \\
&= \langle c_n s_{n-1} \rangle \cdot 2^{n-1} + \langle s_{n-2} \dots s_0 \rangle && \text{Lemma 1.3.1} \\
&= \langle c_n s_{n-1} \cdots s_0 \rangle && \text{Lemma 1.1.2}
\end{aligned}
$$

■

Note that the sum contains $n + 1$ bits. If the carry bit out of the last position is 0, Lemma 1.3.5 tell us that the lower n bits are accurate, i.e. $\langle s_{n-1} \dots s_0 \rangle = \langle a \rangle + \langle b \rangle$. If the carry out of the last position is 1, the lower n bits are not accurate and the sum is greater or equal 2^n. In this case an **overflow** happened and we would need all $n + 1$ bits to accurately represent the result. Since computers internally operate on bit strings of fixed size, there is no space to store this additional bit directly in-place with the result. If we are only interested in the n least significant bits when adding, we write shortly

$$a +_n b \coloneqq s_{n-1} \dots s_0 \tag{1.3}$$

to indicate that we ignore the carry out of the most significant place.

Let us now turn to subtraction. Instead of inventing a new algorithm for subtracting, we are going to use a trick to express subtraction by addition. This trick hinges on the fact that we are only interested in subtracting words of n bits and not bit strings of arbitrary length. Let us first understand the overall idea using an example in base 10 arithmetic:

Assume we want to subtract base 10 numbers of length 3, and more concretely suppose we want to compute $3 - 2$.

$$3 - 2 \equiv 3 - 2 + 1000 \equiv 3 + 998 \equiv 1001 \equiv 1 \mod 1000$$

What essentially happens here is that we, instead of subtracting 2, we added $998 = 1000 - 2$ and just considered the result modulo 1000 which is 1, the desired result. This of course only works if we limit the digit sequences to a certain length before (here 3).

We use the same trick to subtract two bit strings of length n and therefore define subtraction to fulfill the following specification:

$$\langle a - b \rangle \coloneqq \langle a \rangle + (2^n - \langle b \rangle) \tag{1.4}$$

So, instead of subtracting $\langle b \rangle$ from $\langle a \rangle$ we are *adding* $(2^n - \langle b \rangle)$ to $\langle a \rangle$. Let us first observe that the result $\langle a \rangle + (2^n - \langle b \rangle)$ is equivalent to $\langle a \rangle - \langle b \rangle$ modulo 2^n analogously to the example above. This means that the lower n bits of $\langle a \rangle + (2^n - \langle b \rangle)$ and lower n bits of the true difference are equal which is exactly what we wanted. The number $2^n - \langle b \rangle$ is called the **two's complement** of $\langle b \rangle$.

We will see in the proof of Lemma 1.1.6 that we can easily compute the two's complement of $\langle b \rangle$ by flipping all bits of b and adding 1. The latter can easily be achieved by setting the carry *into* the least significant position (c_0) to 1. Additionally, we see that if $\langle a \rangle \geq \langle b \rangle$ (i.e. the result of the subtraction is defined), $\langle a - b \rangle$ will be greater than 2^n. Hence, the carry *out* of the most significant bit is *set* if the result of the subtraction is valid and is *clear* when the subtraction overflowed.

Definition 1.3.6 Subtraction. $a - b \coloneqq a + \overline{b}$ with $c_0 = 1$ and $a -_n b \coloneqq a +_n \overline{b}$ with $c_0 = 1$ ◊

Lemma 1.3.7 Subtraction is sound. $\langle a - b \rangle = \langle a \rangle + 2^n - \langle b \rangle$ *(i.e.* (1.4) *holds)*

Proof.

$$\langle a - b \rangle = \langle a + \overline{b} \rangle \text{ with } c_0 = 1 \qquad \text{Definition 1.3.6}$$

$$
\begin{aligned}
&= \langle a \rangle + \langle \overline{b} \rangle + 1 && \text{Lemma 1.3.5} \\
&= \langle a \rangle + \left(\sum_{i=0}^{n-1} (1 - b_i) 2^i \right) + 1 && \text{Definition 1.1.1} \\
&= \langle a \rangle + \left(\sum_{i=0}^{n-1} 2^i \right) + 1 - \left(\sum_{i=0}^{n-1} b_i 2^i \right) \\
&= \langle a \rangle + \left(\sum_{i=0}^{n-1} 2^i \right) + 1 - \langle b \rangle && \text{Definition 1.1.1} \\
&= \langle a \rangle + 2^n - \langle b \rangle && \text{Lemma 1.1.6}
\end{aligned}
$$

■

Remark 1.3.8 Practical Considerations. Definition 1.3.6 already suggests how an adder has to be extended to support subtraction: The right operand of the ALU must be optionally negateable and the carry bit into the least position must be presettable to either 0 or 1.

Many processors provide the carry out of the last position in a special register, the so-called **flag register**. It is then often just called the *carry bit* because all the intermediate carries are unaccessible to the programmer. Some architectures negate the carry bit when subtracting, actually making it a *borrow bit*. For example, on an x86 machine the carry bit is *clear* when computing $3 - 2$ whereas an ARM processor would set the carry in that case.

Pro comment: This is why on x86 the instruction that subtracts and reading c_0 from the flags is called `sbb` (sub with borrow) while it is called `sbc` (sub with carry) on ARM processors. Such instructions are used when implementing arithmetic on bit strings that are longer than the word size of the processors.

1.4 The Integers

Up to now we were only assigning non-negative numbers to bit strings using the function $\langle \cdot \rangle$. However, it would be nice to be able to do meaningful computations on bit strings that represent negative numbers as well. To do this, we need to come up with a new interpretation of bit strings that represents positive and negative numbers.

Definition 1.4.1 Signed interpretation of bit strings.

$$[\cdot] : \mathbb{B}^n \to \mathbb{Z} \qquad [b_{n-1} b_{n-2} \dots b_0] \mapsto -b_{n-1} \cdot 2^{n-1} + \langle b_{n-2} \dots b_0 \rangle = \langle b \rangle - b_{n-1} 2^n$$

◊

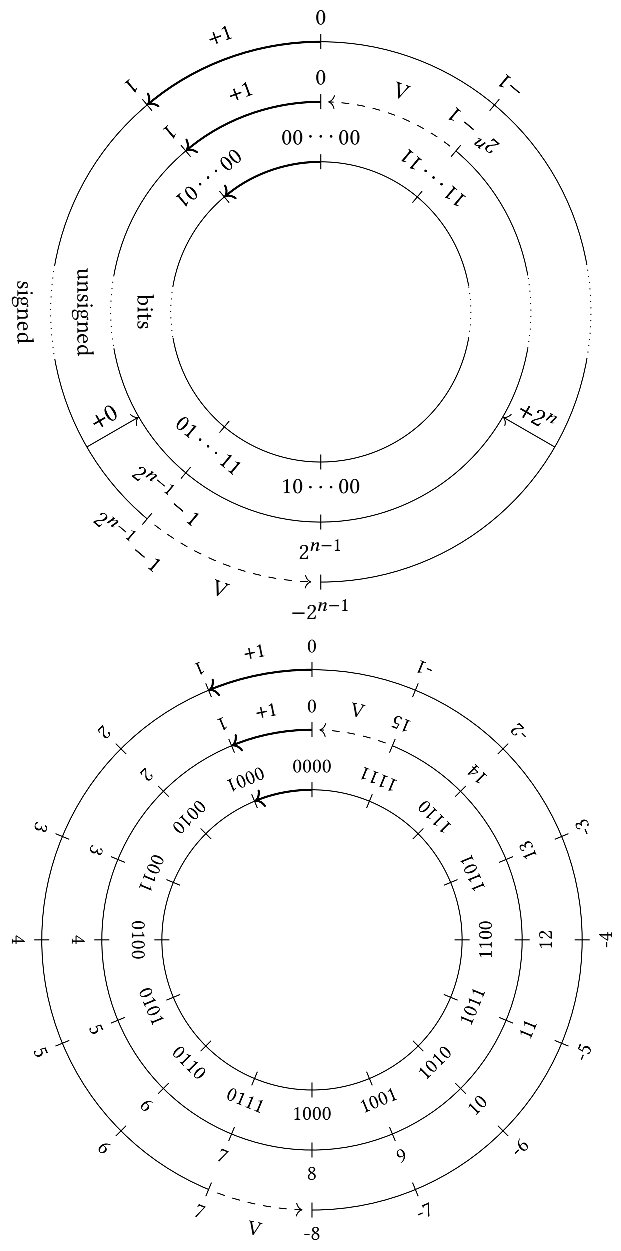


Figure 1.4.2 Bit strings of arbitrary length and of length 4 and their respective unsigned and signed numbers. V indicates overflow.

We have proven (Lemma 1.3.5) that the addition algorithm of Definition 1.3.3 always preserves the values under the unsigned interpretation *modulo* 2^n. We have already discussed the role of the carry out of the most significant position to detect overflows in the case of adding or subtracting unsigned numbers. We will see shortly that our addition algorithm also works for signed numbers in a similar way: It preserves the signed interpretation modulo 2^n but has a different criterion for determining an overflow, as the following examples show:

$$\langle 011\rangle + \langle 110\rangle = 9 \neq \langle 011 +_3 110\rangle = 1$$
$$[011] + [110] = 1 = [011 +_3 110]$$
$$\langle 011\rangle + \langle 001\rangle = 4 = \langle 011 +_3 001\rangle$$
$$[011] + [001] = 4 \neq [011 +_3 001] = -4$$

The next lemma clarifies under which circumstances the lower n bits of an addition represent the accurate result with respect to the signed interpretation and also gives a condition for the overflow:

Lemma 1.4.3 Signed addition. $[a] + [b] = [a +_n b]$ *if and only if* $c_n = c_{n-1}$.

Proof.

$$\begin{aligned}
[a] + [b] &= -(a_{n-1} + b_{n-1}) \cdot 2^n + \langle a \rangle + \langle b \rangle && \text{Definition 1.4.1} \\
&= -(a_{n-1} + b_{n-1}) \cdot 2^n + \langle a + b \rangle && \text{Lemma 1.3.5} \\
&= -(a_{n-1} + b_{n-1}) \cdot 2^n + \langle c_n s_{n-1} \rangle 2^{n-1} + \langle s_{n-2} \dots s_0 \rangle && \text{Lemma 1.1.2} \\
&= (c_n - a_{n-1} - b_{n-1}) \cdot 2^n + s_{n-1} 2^{n-1} + \langle s_{n-2} \dots s_0 \rangle && \text{Lemma 1.1.2} \\
&= (c_n + s_{n-1} - a_{n-1} - b_{n-1}) \cdot 2^n + [s_{n-1} \dots s_0] \\
&= (c_n + a_{n-1} + b_{n-1} + c_{n-1} - 2c_n - a_{n-1} - b_{n-1}) \cdot 2^n + [a +_n b] && \text{Lemma 1.3.1} \\
&= (c_{n-1} - c_n) \cdot 2^n + [a +_n b]
\end{aligned}$$

■

Remark 1.4.4 Practical Considerations. As mentioned before, many processors provide the carry out of the most significant position as the *carry flag* in a flag register. Additionally, these processors typically provide an **overflow bit** which is just the xor between the carry into and out of the most significant position as indicated by Lemma 1.4.3.

Checkpoint 1.4.5 Overflow Detection for Signed Subtraction. Show that overflow detection for signed subtraction is equivalent to Lemma 1.4.3

Hint. Adapt Lemma 1.4.3 accordingly.

Checkpoint 1.4.6 Another Way to Check for Signed Overflows. Suppose your ALU does not provide you with an overflow bit and suppose you do not have access to c_{n-1}. So you cannot use Lemma 1.4.3 to compute the overflow bit. Explore how you can compute the overflow bit from the most significant bits of the added numbers, of the sum, and maybe from the carry bit. Come up with adequate boolean expressions that compute the overflow bit from these components.

Hint. Use the value table from Lemma 1.3.1

Answer.

$$v = (a_{n-1} \mathbin{\&} b_{n-1} \mathbin{\&} c_{n-1}) \mid (\overline{a}_{n-1} \mathbin{\&} \overline{b}_{n-1} \mathbin{\&} \overline{c}_{n-1})$$

or alternatively

$$v = (a_{n-1} \mathbin{\&} b_{n-1} \mathbin{\&} \overline{s}_{n-1}) \mid (\overline{a}_{n-1} \mathbin{\&} \overline{b}_{n-1} \mathbin{\&} s_{n-1})$$

Remark 1.4.7 Not every operation gives meaningful results on signed and unsigned alike. Consider the *less-than* operation $a < b$ that yields 1 if $a < b$ and 0 otherwise. For example:

$$\langle 001 \rangle = 1 < 6 = \langle 110 \rangle \quad \text{but} \quad [001] = 1 \not< -2 = [110]$$

So, when performing a comparison on two bit strings, one has to decide if the bit strings are to be interpreted as signed or unsigned numbers. Consequently, modern microprocessors have *two* operations for comparing integers. On MIPS, which we will discuss in the next section, there is `slt` which interprets the bit strings as signed integers and `sltu` which interprets them as unsigned integers.

Remark 1.4.8 Being signed or unsigned is *not* a property of the bit string itself but how you interpret it. A binary operation on bit strings just produces another bit string. It may, however, be sound with respect to the signed or the unsigned interpretation (comparison) or with respect to both (addition).

Sign and Zero Extension. It happens frequently that we want to convert a bit string of length m into a longer bit string of length n so that the longer bit string represents the same number as the shorter one.

Aside: This happens for example if we want to load a single byte from memory but the computer internally works on bit strings of 32 bits.

In such a situation it is important how we want to interpret the bit string: as a signed or an unsigned number because for either case the conversion is different. Converting a shorter to longer bit sequence such that their value stays the same under the unsigned interpretation is called **zero extension** where as **sign extension** preserves their value under the signed interpretation.

b	$[b]$	$\langle b\rangle$
0000	0	0
⋮	⋮	⋮
0111	7	7
1000	−8	8
⋮	⋮	⋮
1111	−1	15

b	$[b]$	$\langle b\rangle$
000	0	0
001	1	1
010	2	2
011	3	3
100	−4	4
101	−3	5
110	−2	6
111	−1	7

b	$[b]$	$\langle b\rangle$
00	0	0
01	1	1
10	−2	2
11	−1	3

Figure 1.4.9 Bit strings of lengths 4, 3, and 2. Preserving the value under the unsigned interpretation means extending with zeros. Preserving the value under the signed interpretation means replicating the most significant bit.

Figure 1.4.9 shows all bit strings of lengths 4, 3, and 2. Apparently, when the longer bit string preserves the value of the shorter under $\langle\cdot\rangle$ the bit string is extended with zeros. In the case where the signed value shall be preserved, the most significant bit is replicated: 11 becomes 111 becomes 1111, whereas 01 becomes 001 and 0001. Proving that zero extension preserves the unsigned value is straightforward. The signed case is slightly more interesting:

Lemma 1.4.10 $[b_{n-1}b_{n-1}\dots b_0] = [b_{n-1}\dots b_0]$

Proof.

$$\begin{aligned}
[b_{n-1}b_{n-1}\dots b_0] &= \langle b_{n-1}\dots b_0\rangle - b_{n-1}\cdot 2^n && \text{Definition 1.4.1}\\
&= \langle b_{n-2}\dots b_0\rangle + b_{n-1}\cdot 2^{n-1} - b_{n-1}\cdot 2^n && \text{Definition 1.1.1}\\
&= \langle b_{n-2}\dots b_0\rangle - b_{n-1}\cdot 2^{n-1} && \text{Definition 1.4.1}\\
&= [b]
\end{aligned}$$

■

1.5 Shifts

It is easy to multiply a bit string by 2^k: You just shift each digit k positions to the left and fill the lower k positions that have been freed with 0. Because we only consider bit strings of a fixed length n, the most significant bit will be lost when shifting left by 1.

Definition 1.5.1 Left Shift.

$$x_{n-1} \ll k := x_{n-1-k} \ldots x_0\, 0^k$$

◊

Dividing by powers of 2 works similarly. Instead of shifting left, we have to shift right. Here, the most significant positions are filled with zeros.

Definition 1.5.2 Unsigned Right Shift.

$$x_{n-1} \gg k := 0^k\, x_{n-1} \ldots x_k$$

◊

However, this does not preserve the sign under the signed interpretation if the bit string denotes a negative number. Consider the following example and let the length of the bit string be 4:

$$[1000] = -8 \quad \text{but} \quad [1000 \gg 1] = [0100] = 4$$

To remedy this problem, we introduce another right shift. The signed right shift does not fill in zeroes but replicates the most significant bit:

Definition 1.5.3 Signed Right Shift.

$$x_{n-1} \overset{s}{\gg} k := x_{n-1}^k\, x_{n-1} \ldots x_k$$

◊

Checkpoint 1.5.4 Show that the shift definitions are sound modulo 2^n. For the left shift, this means proving $\langle x \rangle \cdot 2 \equiv \langle x \ll 1 \rangle \mod 2^n$. Formulate respective criteria for the two right shifts and prove them as well.

1.6 Summary

In this chapter, we covered the basics of computer arithmetic for integers. We have discussed two interpretation functions that assign bit strings natural numbers and integers. One crucial observation is that bit strings are not per se signed or unsigned but the algorithms we employ on them have to be sound with respect to the one or the other interpretation.

We have introduced a simple addition algorithm that uses elementary boolean functions (and, or, xor) and can easily implemented in hardware. Furthermore, we have seen how subtraction can easily implemented based on addition by complementing the right operand and setting the carry into the least position to 1.

One crucial limitation in computers is that they intrinsically operate on word-size data and we have seen that it is not always possible to accurately represent the result of an addition because adding to n-bit numbers may require $n + 1$ bits for the result. We have discussed under which circumstances the least significant n bits of the addition result are exact, i.e. no overflow occured. For unsigned addition and subtraction the carry out of the last position gives an indication for an overflow. For signed addition and subtraction, the xor of the carries into and out of the last position detects an overflow.

Finally, we have briefly presented sign- and zero-extension of bit strings to convert bit strings into longer bit strings while maintaining their signed/unsigned interpretations and we have briefly looked into shifts which can be used to implement multiplication and division by powers of two.

Basic knowledge about computer arithmetic is important, because signed and unsigned numbers pop up in the type systems of many statically-typed imperative programming languages such as C, Java, and so on.

Chapter 2

Machine Code

We start with a very simple imperative programming language, the machine code for the MIPS instruction set architecture. An **instruction set architecture** is the definition of a machine code language. There may be various different *implementations* (i.e. different processors) that all implement the same ISA. For example, all modern ARM processors from different vendors implement the AArch64 ISA.

Machine code is very primitive, does not contain a lot of concepts, and is very close to the machine. By learning machine code, we get an understanding of the lowest abstraction in programming, the so-called **hardware-software interface**, which is essentially how software "talks" to hardware. Understanding what happens at the machine code level is relevant in many situations: For example, many software security problems exploit the fact that some programming languages cannot really abstract the machine code level entirely which can be exploited to "make the program behave" in an unintended way. Another example is performance optimization: If code needs to run as fast as possible, engineers often look at the machine code generated from a compiler to understand possible performance bottlenecks. Sometimes performance-critical code is even written in machine code.

The abstractions that higher programming languages (such as C or Java) provide are of course helpful because they save us work. They relieve the programmer from having to think about low-level details and enable *portable code*. Portable means that a program can be run on different machines that have possibly different instruction set architectures. One way of enabling portability is using a **compiler**. A compiler is a program that translates a program written in one language (e.g. C) to a program in another language (such as machine code).

2.1 The von-Neumann Architecture

The foundations of computer architecture have been laid in the late 1940s by the hungarian-american mathematician John von Neumann in his groundbreaking report on the EDVAC computer The basic architecture of computers hasn't changed fundamentally since then.

A computer consists of several different components (CPU, memory, input/output) that all communicate via a shared bus. The bus serves as the central means of communication that connects all components of the system. It is just a "bunch of wires" that can be accessed (written to and read from) by all components. Busses essentially avoid the overhead of n:n connections between the components at the price of the fact that only one bus participant can write to the bus at one point in time.

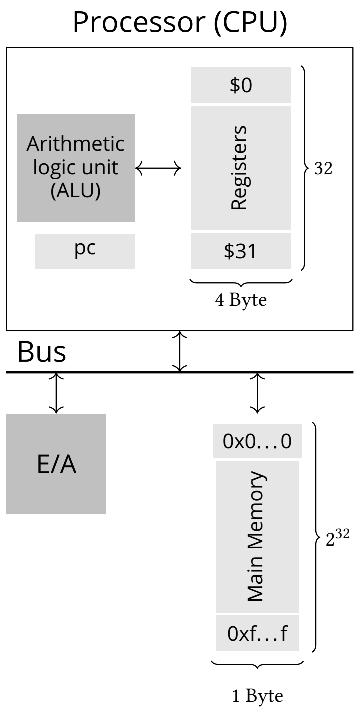


Figure 2.1.1 High-level schematic overview of a computer with a MIPS processor

A core abstraction that is common in most computers is that of addressable memory. The system has an **address space**, i.e. a range of natural numbers that each stands for the **address** by which a piece of data (commonly a byte) can be accessed. That data does not necessarily has to be stored in one single memory. Input-output devices can blend in parts of their memory in the global address space to make it accessible by all bus participants. The **central processing unit (CPU)** where all computations take place is also just a bus participant. It obtains and provides the data it computes on via the bus to the rest of the system.

2.2 The MIPS Processor

Aside: MIPS was one of the early RISC processor designs that emerged in the early 1980s. Its instruction set is very uniform and simple and therefore well-suited for an introduction to machine code programming. It is the spiritual ancestor to the RISC-V instruction set architecture which has gained significant traction in recent years. For the sake of simplicity, we are considering one of the earlier, 32-bit ISA definitions because they are less complex than later ISA extensions.

In this section, we introduce the instruction set architecture (ISA) of the **MIPS processor**. Figure 2.1.1 shows a conceptual overview over a computer with a MIPS processor. Of course, a concrete implementation is much more complex but we are focussing on the parts that are relevant for learning machine code programming. So from this very high-level perspective, a MIPS processor contains an **arithmetic logical unit (ALU)** that carries out all the computations. It also contains a **register file**, a small but fast storage facility where the programmer can store 32 values of 32 bits each. Register 0 cannot be written to and always reads 0. This is a “trick” to provide the constant 0 that is frequently used. Like all the other bus participants, the processor is connected to the bus via which it can send and receive data to the rest of the system, including the main memory. The address space of the system we are looking at here is 32-bit, which means that it contains the addresses from 0 to $2^{32} - 1$.

Remark 2.2.1 Registers/Memory. Why do we need registers if we have main memory to store data? There mainly two reasons for having registers:

1. *Speed.*

   Retrieving data from main memory is quite slow in comparison to the speed of the processor. For technical reasons, it can take several dozens to hundreds of clock cycles. So it is unpractical for a CPU to retrieve the data it operates on from the main memory constantly.

2. *Addressability.*

   The address of a datum in main memory is 32-bits wide. If the CPU operated on main memory directly, a simple addition would need to invest three times 32 bits to describe the operands and the target memory cell which sums up to 96 bits. This makes the instruction word quite large in comparison to the three times 5 bits needed to address the register file.

Remark 2.2.2 Caches. If instructions are fetched from main memory and main memory accesses take so long compared to the speed of the processor, you may wonder why the processor's speed is actually higher than the one of main memory. The reason is that concrete systems use a hierarchy of so-called cache memories between the CPU and the actual main memory. These caches trade-off speed versus size and are able to provide data quicker, cache recently-accessed memory cells and also prefetch data from lower hierarchies that is “likely” to be used next. This has the effect that the instructions that will be fetched in the near future reside very closely to the CPU in the cache hierarchy.

Caches are mostly transparent to the programmer. We only need to care about them when we want to optimize our program for speed or are concerned with multi-processor systems that can execute several *threads* of a program simultaneously and therefore need to share data consistently. Both topics are out of the scope of this chapter.

A processor executes instructions. These instructions are just data that is stored in the memory of the computer. The **program counter (PC)** is a register inside the CPU that contains the address of the instruction that is currently executed. In MIPS (and many other modern ISAs), each **instruction word** is 32 bits wide. The vast majority of instructions fall in one of the following three categories:

1. *Computation.*

   These instructions take two operands of which one is read from the register file and the other either also from the register file or directly from the instruction word. In the latter case, the operand is called an **immediate**. Immediates allow for embedding constants into the instruction word which is needed quite often when programming (think of adding 1 to a number for example). The **arithmetic-logical unit (ALU)** performs all the operations we have discussed in Chapter 1 (addition, subtraction, and, or, xor, and more) and writes the result back to the register file. The MIPS processor we look at has a **word size** of 32 bit which means that the ALU inputs and outputs are 32 bits wide.

2. *Memory Access.*

   These instructions compute an address based on the contents of a register and an optional immediate and use this address to either load or store data from or to the main memory.

3. *Control Flow.*

   When the processor executes a computation or memory instruction, it increments the program counter by 4 to let it point to the instruction right after the one that is currently been executed. Control flow instructions can influence the value of the program counter in various different ways that we discuss below. Essentially, they serve the purpose to alter the program's flow which is important to implement repeated and conditional execution of code. Without that, computers would not be very powerful because they only could execute a simple list of instructions.

Instruction Encoding. Let us look at an example of how instruction words look like. As mentioned before, an instruction is just a 32-bit word. The bits in the word encode the operation that the processor shall carry out, the registers it accesses, and potential immediates. Let us consider the word `0x00641021`. It consists of individual bit fields that represent the different properties mentioned above.

$$\text{0x00641021} = \underbrace{000000}_{\text{opcode}}\ \underbrace{00011}_{\text{rs}}\ \underbrace{00100}_{\text{rt}}\ \underbrace{00010}_{\text{rd}}\ \underbrace{00000}_{\text{unused}}\ \underbrace{100001}_{\text{funct}}$$

$$= \begin{array}{|c|c|c|c|c|c|}\hline 0 & 3 & 4 & 2 & 0 & 33 \\ \hline \multicolumn{1}{c}{6} & \multicolumn{1}{c}{5} & \multicolumn{1}{c}{5} & \multicolumn{1}{c}{5} & \multicolumn{1}{c}{5} & \multicolumn{1}{c}{6} \end{array}$$

opcode, funct	Specifies the operation to be carried out. Here, opcode=0 funct=33 means that the ALU shall add the numbers.
rs, rt, rd	Specifies the registers of the first and second operand and the register where the result shall be placed.

2.3 The Assembler

We have seen in the last section that the instructions that a processor executes are just bit strings that reside in memory. Because we do not want to encode these bit strings by hand, we use an **assembler**. An assembler translates a textual representation of machine code into **binary code**, i.e. the bit strings of the instructions. To this end, the assembler reads in a file that contains machine code instructions in their textual form and **directives** for the assembler that direct the assembly process and influence the shape of the output. The output of the assembler is a **binary file** (sometimes called **object file**). This file contains the binary-encoded machine code instructions, data for the data segment, and meta-data for the **linker** that binds together multiple object files into an executable program. Figure 2.3.1 shows this process schematically. As indicated in Figure 2.3.1 a program can consist of multiple assembly files. Each such file is called a **translation unit** because they are all "translated" separately.

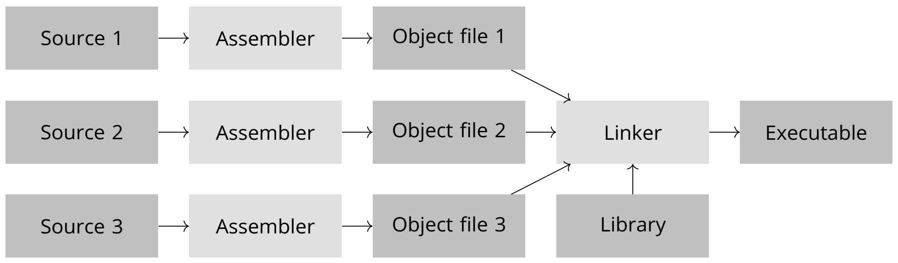


Figure 2.3.1 Translation process for an assembly code. The assembler translates each source file to object code and the linker binds them together to form an executable.

2.3.1 Overview

Let us explore the ingredients of an assembly code file by means of the example in Listing 2.3.2. The figure shows six instruction words that are located somewhere in the memory of the computer. The left column shows the instruction words as hexadecimal 32-bit numbers. The second and third column show the textual representation of the respective instruction word.

```
24020001 addiu $2 $0  1
04010002 bgez  $0 2
70441002 mul   $2 $2 $4
2484ffff addiu $4 $4 -1
1c80fffd bgtz  $4 -3
03e00008 jr    $31
```

Listing 2.3.2 Instruction words in hex and assembler instructions of a function that computes the factorial of a number given in register 4.

The second column represents the so-called **mnemonic** which is a short textual description of the opcode, i.e. the operation the instruction stands for such as `mul` for multiplication or `addiu` for “add a register to an immediate and ignore overflows”. The last column gives the operands of instruction. Registers are prefixed by the $ sign, e.g. $2 refers to register 2. Numbers without dollar signs represent immediates. So, `addiu $2 $0 1` adds the value 1 to the contents of register 0 and stores the result in register 2. Effectively, this instruction places the value 1 into register 2 because register 0 always reads as zero. The instructions `bgtz` and `bgez` are control flow instructions that alter the value of the program counter. `bgtz $4 -3` for example tests, if the value in register 4 is greater than zero (=gtz). If this is the case, the program counter is advanced by -3 instructions relative to the next instruction. (so, if the branch instruction is located at address 0x04000008, the pc will be set to 0x04000000.) If this is not the case, the program counter advances normally to the next instruction and the branch has no other effect. When adding a value to an address (like adding -3 to the pc in this example), this value is called an **offset**. Branches that take offsets are called **relative branches** whereas branches that take addresses as operands are called **absolute branches**. `jr` is an absolute branch: it sets the program counter to the contents of its operand register. A branch instruction that branches on a condition like in this example, is called a **conditional branch**. Sometimes one also dedicates the term branch specifically to conditional branches and calls unconditional branches **jumps**. The instruction `bgez $0 2` acts as an **unconditional branch** here. `bgez` checks if the operand register is greater or equal than zero (=gez). Since the operand register here is register 0 which is always zero, the condition is always fulfilled and the instruction will branch *every time* it is executed making it an unconditional branch instruction, effectively.

Remark 2.3.3 Pseudo instructions. It may seem as a crude "hack" to use conditional branches to branch unconditionally. However, it saves us from introducing a dedicated instruction for that. This way, the instruction set stays small and clean which is part of the **RISC** (reduced instruction set) paradigm. The assembler however offers **pseudo instructions** such as `li` (load immediate = put an immediate into a register) or `b` (branch) that are just abbreviations for commonly used operations that can be expressed with the actual instruction set in a not so straightforward way.

2.3.2 An Example

Computing offsets for relative branches is tedious and error-prone. Even worse, if one inserts new code between the branch and its target, the offset has to be recalculated. One of the prime tasks of an assembler is therefore to provide **labels**. Every entity in an assembler file that will be put somewhere in memory (such as instructions but also static data, Section 2.5 Every instruction (in general every address in a segment) can be marked with a label that stands for the address of the entity. One can refer to these labels at various places (such as in the operand list of relative branch instructions) and the assembler automatically computes the appropriate offsets for us. Listing 2.3.4 shows the assembly file from which the assembler produced the binary code shown in Listing 2.3.2.

```
    .text
    .globl factorial
factorial:
    li    $v0 1       # load 1 into register $v0
    b     check       # jump to label check
loop:
    mul   $v0 $v0 $a0 # multiply $v0 with $a0 and put
                      # result into $v0
    addiu $a0 $a0 -1  # Subtract 1 from $a0
check:
    bgtz  $a0 loop    # branch to loop, if $a0 > 0
    jr    $ra         # return to caller
```

Open in Browser

Listing 2.3.4 A function that calculates the factorial of a number. This number is expected to be in register `$a0` when the function is invoked.

Labels are defined by giving a name followed by a colon. `factorial`, `loop`, `check` are all labels. All other occurrences of these labels refer to them. The address the label stands for is the address at which the instruction that follows the label will be placed in memory when the program is loaded.

Another difference from the assembly code in Listing 2.3.2 to Listing 2.3.4 is that it is customary to use short names for registers. Instead of referring to register 4 as $4 we write `$a0` here. This short name comes from the so-called **calling convention**, a set of rules the *we* use to assign specific roles to registers when calling function which we will discuss in Section 2.8.

Furthermore, Listing 2.3.4 shows some examples for assembler directives. `.text` is a directive that indicates that everything that follows is code. It activates the assembly code function which allows us to write mnemonic and register names instead of hand-coding instructions. `.text` also tells the linker later on that everything that follows has to be placed into the code (sometimes called *text*) segment. We will discuss segments later in Section 2.6. `.globl factorial` makes the label `factorial` visible from other translation units (see Figure 2.3.1. Labels that are not declared global are local to a translation unit and cannot be referred to from other translation units. This prevents that situation that programmers accidentally use the same label name in different files that would clash if the label's name was global. Note also the use of pseudo instructions as discussed in Remark 2.3.3.

Finally, let us give a main program that calls our factorial function. By convention, program execution starts at instruction with the label `main`. Our main program shall compute that factorial of 10, display the output on the console, and terminate the program.

```
    .text
    .globl main
main:
    li   $a0 10      # load 10 into register $a0
    jal  factorial # call function factorial
    move $a0 $v0    # result is in $v0; move to $a0
    li   $v0 1      # load syscall number 1 into $v0
    syscall         # call system to output
    li   $v0 10     # load syscall number 10 into $v0
    syscall         # call system to end program
```

Open in Browser

Listing 2.3.5 A main program for our factorial function

Our main program calls the operating system to perform input and output operations. Here, we do go into details about operating systems. For our purposes, it is sufficient to accept that there is some system that we can interact with using the `syscall` instruction. We specify what exactly we want the operating system to do by putting a certain number into the `$v0` register. Here, 1 means print the number stored in `$a0` as a decimal number on the console and 10 means terminate the program.

Remark 2.3.6 Program Termination. Why do we explicitly need to terminate the program with a syscall? Listing 2.3.5 suggests that it is clear that the program ends after the last instruction. However, the instructions of our program are just some bytes in memory and behind these bytes there are other bytes that don't belong to our program. So our processor doesn't "see" the end of our program like we do in the listing above. Therefore, we have to hand back control to the operating system when our program has ended.

There is one other new instruction in Listing 2.3.5: `jal`. `jal` stands for "jump and link". A `jal` instruction has an immediate that it interprets as an absolute address [3] before setting the program counter to that address, it stores the address of the instruction that follows the `jal` into register 31. This is the address the function that `jal` calls will want to return to. This is why register 31 is also called `$ra` (= return address).

2.4 Execution Traces

To develop an intuition for what happens during the execution of a program, it is helpful to write down an execution trace of the program. An execution trace lists the instructions in the order they are executed and shows the effect of each instruction. In imperative programming languages (like our assembly language here), the execution of a program consists of multiple (potentially infinitely many) steps. In every step one instruction is being executed. The effect of an instruction is either a change of

[3] The immediate is 26 bits wide. The final address is formed shifting that immediate 2 to the left (each instruction is 4 bytes long, so we never want to jump to an address that is *not divisible* by 4) and then prepending this with the 4 upper bits from the address where the jump instruction is located. So a MIPS jump instruction can only jump to an absolute address inside a 256 MiByte region.

the **state** of the machine (changing the contents of a register or a memory cell) or an **external effect** which is an interaction with the environment (printing to the console, reading from a file, etc.).

Remark 2.4.1 State. The structure of the state is a part of the semantics of the language. In the MIPS machine code language, the state is given by the contents of the registers and the memory. When discussing higher-level programming languages later, we introduce the concept of a variable which will make our states more "structured"

Let us reconsider the factorial program from the introduction and let us assume the first instruction is located at address 0x00400000. This is the typical address at which the operating system loads our MIPS programs. So the first couple of bytes in memory starting from this address contain our program which is indicated by this **disassembly**:

```
0x00400000: addiu $v0 $0 1
0x00400004: bgez  $0 2
0x00400008: mul   $v0 $v0 $a0
0x0040000c: addiu $a0 $a0 -1
0x00400010: bgtz  $a0 -3
0x00400014: jr    $ra
```

We write down an execution trace as a table. Each row is the snapshot of the state *before* executing an instruction. Each column shows one part of the state that is of interest. Note that the address of the instruction that is to be executed next is also part of the state. The following table shows the execution trace for our example program with a start state in which register `$a0` contains the value 3 (of course it contains a bit string of 32 bits whose unsigned and signed interpretation is 3).

`$pc`	`$v0`	`$a0`
0x00400000	?	3
0x00400004	1	3
0x00400010	1	3
0x00400008	1	3
0x0040000c	3	3
0x00400010	3	2
0x00400008	3	2
0x0040000c	6	2
0x00400010	6	1
0x00400008	6	1
0x0040000c	6	1
0x00400010	6	0
0x00400014	6	0

To determine the row in the trace, we need to know what effect the instruction at the current position has. To this end, Appendix A summarizes the effects of the MIPS instructions we are going to use.

At the beginning, the program counter contains address 0x00400000. The instruction at this address is `addiu $v0 $0 1`. This instruction adds the constant 1 to the contents of register `$0` and puts the result into register `$v0`. (Of course, it also increases the program counter by 4.) Therefore, the next instruction to execute is located at address 0x00400004. This instruction, `bgez $0 2`, branches 2+1 instructions ahead if the content of register `$0` equals 0, which it always does. In fact, this instruction is an unconditional jump that sets the program counter 3 instructions ahead and so on and so forth. We'll stop the execution trace for this example when reaching the `jr $ra` instruction that would return to the caller of this function. (More on functions later in Section 2.8).

2.5 The Data Segment

In addition to the code, programs also contain **static data**. The term “static” means that the extent (the size) of the data is known *statically* which means it is known prior to the program’s execution and does not vary during execution. This means that the memory for static data can be provisioned during assembly and linking and allocated directly when the program is loaded into memory. As we will see later in Section 2.6, the memory the program uses is organized into segments that each serve a distinct purpose. The **data segment** is the segment that holds the static data.

Remark 2.5.1 Statically versus Dynamically Allocated Memory. The fact that there is statically-allocated memory indicates that there is also dynamically-allocated memory. We do only touch on this in this chapter but will discuss dynamic memory allocation in more detail when we talk about C programming. A large part of the memory programs use is allocated dynamically, i.e. during the runtime of the program. The reason is that in most situations the input to the program influences the amount of memory that has to be allocated, so there is no way of knowing *statically* (i.e. before the program runs) how much memory we need. Consider, for example, a program that processes an image that the user can select after starting the program. We do not know the size of that image statically.

It is common that static data is initialized, i.e. it is set to certain values before the program is started. A typical example for such data are strings (character sequences, i.e. text) that the program uses. If a program prints out a message, that message has to be kept somewhere.

In the assembler file, the `.data` directive starts the declaration of static data. Use `.space n` to reserve some uninitialized n bytes. To reference this memory later on, you can put a label in front of the directive like so:

```
    .data
some_bytes:
    .space 1000
```

To make allocating static data more comfortable, there are a couple of directives to create static data of different sizes and initialize it at the same time. The directives `.byte`, `.half`, `.word` allocate bytes, half-words (MIPS slang for 2 bytes), and words (4 bytes). `.ascii` s and `.asciiz` s allocate memory that is initialized to the ASCII codes of the string s. `.asciiz` also appends a byte with value 0. This is a common way of signaling the end of the string that is used, for example, in the language C. For example:

```
    .data
hello:
    .asciiz "Hello␣World"
```

corresponds to the directives

```
    .data
hello:
    .byte 0x48, 0x65, 0x6c, 0x6c, 0x6f, 0x20
    .byte 0x57, 0x6f, 0x72, 0x6c, 0x64, 0x00
```

One peculiar aspect is that data must be **aligned** depending on its size. If the processor accesses n bytes of memory using a load or store instruction, the address of access needs to be divisible by n, otherwise the processor will trigger an exception and stop executing. For example, the following program

```
    .data
    .ascii "Hallo"
x:
    .byte 8, 0, 0, 0

    .text
    .globl main
main:
    lw $t0 x
```

would cause an exception because the address of label `x` is 0x10000005 which is not divisible by 4. But since we are using the `lw` instruction that loads a word (4 bytes) into a register, we get an exception. We can force a specific alignment for the next label using the `.align` directive like so:

```
    .data
    .ascii "Hallo"
    .align 2
x:
    .byte 8, 0, 0, 0

    .text
    .globl main
main:
    lw $t0 x
```

Open in Browser

Remark 2.5.2 Practice. In practice, the data segment is further subdivided. First, there is a division of read-only data, for example for string constants. The operating system ensures (using virtual memory, which do not go into here) that writing to read-only segments causes an exception and leads to the termination of the program. Second, data that is initialized to 0 is specifically marked and not allocated space in the binary file of the program. Instead, the operating system initializes such memory to 0 when the program is loaded.

2.6 Memory Segmentation

When the user starts the program, the operating system loads it into the memory. Each section of the binary (code, data) will be put at specific addresses in the address space of the program. When running MIPS programs and tracing their execution on the lowest level, it is important to know where in memory which part of the program is located. Note that modern machines provide the abstraction of virtual memory which makes it possible that the each program that runs on the machine uses its own address space. The processor, in cooperation with the operating system, takes care of mapping these virtual addresses to the physical ones by a technique called *address translation*. Virtual memory is beyond the scope of this book. Here, we do not make the distinction of virtual vs. physical memory and only consider a single program at the time.

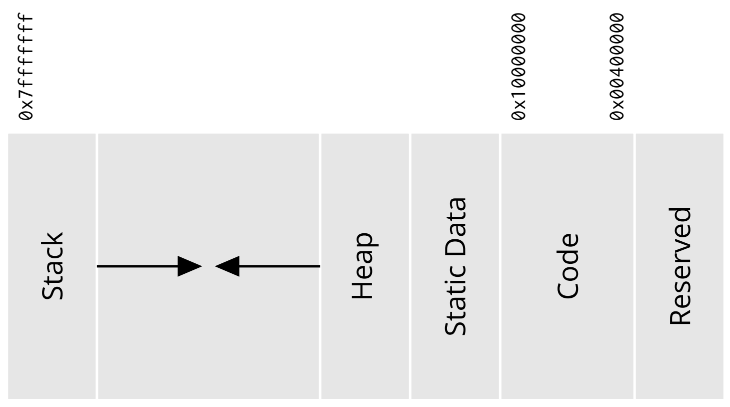


Figure 2.6.1 Typical memory segments in a binary

In our setting, code starts at address 0x00400000 and the data segment starts at 0x10000000. The addresses above 0x10000000 + n, where n is the size of the static data, is called the **heap**. The heap can be used freely by the program (see below). The upper 2GB of the address space (starting from address 0x80000000) are reserved for the operating system. Right below starts the **stack**. The stack is used to hold **stack frames** which are used to keep temporary values during a functions call. Essentially, the stack is also an area where memory is allocated dynamically, however with a specific policy that we will discuss in further detail in Section 2.8. Note that the stack grows downwards.

Memory Management on the Heap. When executing machine code that has been produced by a compiler from a higher-level programming language, the heap is usually managed by the **runtime system** of the language. The runtime system of a language implements some of the abstractions the language provides, for example dynamic memory allocation. When programming machine code directly, there is no runtime system and we have to manage the heap ourselves.

Note that because of virtual memory, we cannot just access memory on the heap because initially there are no physical memory pages mapped into our virtual address space beyond this point. We need to tell the operating system before that we want to do so. The operating system then takes care of providing us with enough (if available) physical memory in our virtual address space. Again, this is beyond the scope of this text and we will just ignore heap memory for the rest of *this chapter*.

A simple memory management system typically provides two functions: One to allocate memory and one to free previously allocated memory. In C they are called `malloc` and `free`. Calling `malloc`, the program can obtain a fresh chunk of memory of a size that the program can specify. `malloc` guarantees that it never returns the address of a chunk that has been malloc'ed before and not been freed yet. Internally, `malloc` organizes the heap in such a way that these guarantees are fulfilled. It also deals with the operating system to map enough physical memory into the heap's address space.

We will talk more about dynamic memory allocation in Section 4.12. One thing that is important to understand is that from the perspective of the processor, dynamic memory allocation does not exist and neither is there a difference between static data or data in dynamically-allocated memory. These are just abstractions that we conceived to better align the organization of the flat address space with the way we want to program our computer.

2.7 Linking

As mentioned in Section 2.3, the linker binds different object files together into one executable. We will give a short overview over linking here because it is also relevant for languages like C/C++ that compile down different translation units of code into separate object files.

The main object of the linker is to give each instruction and element of the data segment a definitive address and resolve references to global labels. As mentioned in Section 2.3 one object file can reference a label in another object file. To be able to resolve global labels, each object file carries a **symbol table** that lists all global labels that the object file defines and references. The linker merges the code and data sections of the individual object files respectively and orders them linearly. Except for code and data from dynamically linked libraries[4], which we ignore here, the address of each datum and each instruction is then fixed. The linker can then **patch** absolute addresses global relative addresses. Patching means that the bytes take the address of a referenced label are overwritten with the final absolute or relative address.

```
    .data
    .globl y
y:  .word 42
m:  .asciiz "Please enter a number: "
    .text
    .globl main
main:
    li    $v0 4
    la    $a0 m
    syscall
    li    $v0 5
    syscall
    move  $a0 $v0
    jal   add_y
    move  $a0 $v0
    li    $v0 1
    syscall
```

Symbol	Properties
y	defined, global
m	defined
main	defined, global
add_y	referenced

Open in Browser

```
    .text
    .globl add_y
add_y:
    lw   $v0 y
    addu $v0 $v0 $a0
    jr   $ra
```

Symbol	Properties
add_y	defined, global
y	referenced

Open in Browser

Figure 2.7.1 Two object files with their assembly source code and the symbol tables of the object files.

2.8 Calling Functions

As soon as programs get larger there is a danger of producing spaghetti code, i.e. writing very large chunks of code that does not have a clear focus but does multiple

[4]libraries that are only linked against when the program is loaded

things all at once. Such code is hard to understand and almost never correct. Therefore, we want to break up programs into smaller modules with clear responsibilities. Each module consists of a set of **functions** and is grouped in one sometimes a few translation units. The benefit of writing a subroutine for each single task is that we can develop, debug, test, and reuse such subroutines individually. In this section, we look at the mechanics of defining subroutines in machine code and making them interact using function calls.

Consider the following piece of code that calls a function bar:

```
    .text
foo:
    addu $t0 $a0 $a1
    jal  bar
    addu $v0 $v0 $t0
    jr   $ra
```

Open in Browser

First of all, jal jumps to the function bar and bar, being a function, will ultimately return to its caller, which is our function foo by executing the instruction jr $ra. Remember that jal puts the address of the instruction behind jal into register $ra before jumping. This allows bar to return to from where it was called.

Let us assume that bar is defined in a different translation unit that we do not make any assumption about. So, how can we know that the value we stored into $t0 with the first addu is still intact? Now you might say that we have to look inside the code of bar to be sure that it doesn't overwrite $t0. This is a bad idea for several reasons: First, if the project is bigger this may mean that you have to look at way more code than your own. Above a certain code size, this is no longer feasible. Second, the implementation of bar may change in the future and so might its usage of $t0. So it is a good idea to make as few assumptions as possible about the inner workings of bar. Pessimistically, we have to assume that bar uses $t0 for its own purposes.

One way to "transport" a value over a function call might be to store the value at some memory address specifically chosen for this purpose. This however does not work for **recursive** functions. A function f is called recursively if, during execution there is a call to f before all previous calls have returned. In this case, we would need multiple such memory cells. The following execution trace illustrates the problem:

```
f           g           f
...
sw  $t0 p
jal g
            ...
            jal f
                        ...
                        sw  $t0 p
                        jal g
                        lw  $t0 p
                        ...
                        jr  $ra
            ...
            jr  $ra
lw  $t0 p
...
jr  $ra
```

f is called a second time before the first call returned. Hence, the second store sw $t0 p would overwrite the saved value of $t0 at p.

So, we need to allocate space in memory *for every call* to save the registers of this particular instance. Because each function returns before its caller, we can organize this memory using a **stack**. A stack is a data structure (a way to organize data) with the policy that data can only be removed from the data structure in the reverse order of their insertion which means that the last in is first out (LIFO). To this end, each function that wants to save register contents allocates a **stack frame** (certain amount of bytes) on the stack where the register contents can be saved. Allocation is simple because it is a stack: One register that holds the current height of the stack is sufficient. In practice this register, called the **stack pointer**, points to the last byte that is currently on the stack. Since the stack grows downwards decreasing the stack pointer by *n* allocates *n* bytes on the stack. On MIPS systems register 29 is assigned the role of the stack pointer and therefore bears the short name `$sp`. The following listing extends the example program above with the allocation and de-allocation of a stack frame and loads and stores to save and restore the values of `$t0` and `$ra`.

```
    .text
foo:
    addiu $sp $sp -8    # decrease $sp, allocate frame
    addu  $t0 $a0 $a1
    sw    $t0 0($sp)    # save $t0
    sw    $ra 4($sp)    # save $ra
    jal   bar
    lw    $ra 4($sp)    # restore $ra
    lw    $t0 0($sp)    # restore $t0
    addu  $v0 $v0 $t0
    addiu $sp $sp 8     # deallocate frame
    jr   $ra
```

Open in Browser

Typically, the size of the stack frame is constant (i.e. the same for all program executions). In some situations, one wants to allocate more memory on the stack, maybe also dependent on some input. In such a case, it may not be *statically* known how deep the stack pointer will move throughout the execution of a procedure's code. This makes using constant offsets in the form of immediates like in the example above impossible. In such a situation, one uses an additional register that is called the **frame pointer**. In MIPS, frame pointers reside in register 30 and are called `$fp`. It is common to set the frame pointer right at the entrance of the procedure by saving the old frame pointer and setting the frame pointer to the current stack pointer value.

The stack frame (see Figure 2.8.1) typically contains the following elements:

- **Arguments**, i.e. values that are passed to the procedure. (The convention in MIPS is that the first four arguments are passed in registers `$a0` to `$a3`, hence their name. Remaining arguments are passed on the stack.)
- Space to store the contents of registers that …
    - the procedure wants to overwrite but the caller expects to be unchanged. These are so-called **callee-save** registers because the *callee* has to make sure the values of these registers are preserved from entering the procedure to its return.
    - might be overwritten by procedures that are called by the procedure. These are so-called **caller-save** registers because the caller needs to preserve their value across a procedure call.
- space for further data local to the procedure that don't fit into a register because there is either no free register available or the data is too big to fit into a 32-bit register.

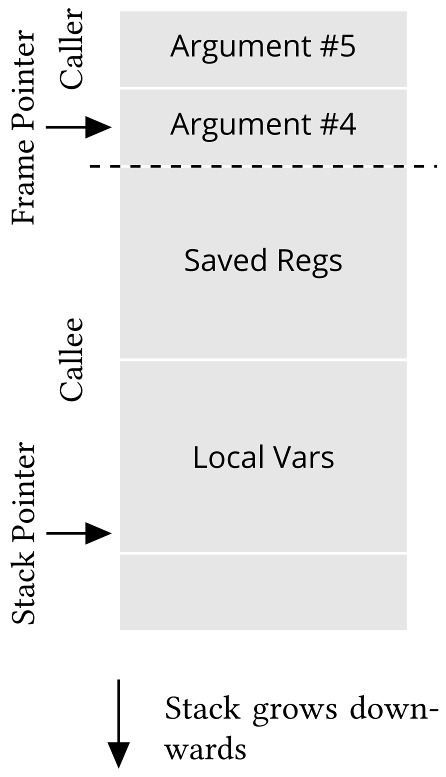


Figure 2.8.1 A stack frame set up when calling a function.

The stack frame is constructed by the caller and the callee together. The caller puts additional arguments on the stack while the callee allocates space to save registers as discussed above. The code of the procedure that builds and tears down the stack frame is called the **prologue** and the **epilogue**.

2.8.1 Recursion

Let us reconsider our initial example, the computation of the factorial function. The standard way of defining the factorial function mathematically is by recursion:

$$f_r(n) := \begin{cases} n \cdot f_r(n-1) & n \geq 1 \\ 1 & n < 1 \end{cases}$$

The implementation in Listing 2.3.4 is not recursive: It does not contain any procedure calls especially not to itself. This implementation is based on a different but equivalent definition of the factorial function which is **tail recursive**

$$f_e(n, r) := \begin{cases} f_e(n-1, n \cdot r) & n \geq 1 \\ r & n < 1 \end{cases}$$

Tail recursive means that the result of the recursive function application *is* the result of the function and does not appear in a larger expression.

Let us look at the implementations of both variants in MIPS assembly shown in Figure 2.8.2

```
                       .text
    .globl fac
fac:
    li      $v0 1
    blez    $a0 end
loop:



    mul     $v0 $v0 $a0
    addiu   $a0 $a0 -1
    bgtz    $a0 loop




end:
    jr      $ra
```

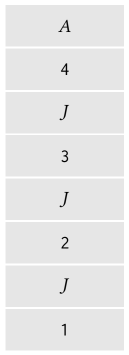
Open in Browser

```
                       .text
    .globl fac_rec
fac_rec:
    li      $v0 1
    blez    $a0 end

    addiu   $sp $sp -8
    sw      $ra 0($sp)
    sw      $a0 4($sp)

    addiu   $a0 $a0 -1
    jal     fac_rec
    lw      $a0 4($sp)
    mul     $v0 $v0 $a0
    lw      $ra 0($sp)
    addiu   $sp $sp 8
end:
    jr      $ra
```

Open in Browser

Figure 2.8.2 Tail-recursive and recursive implementation of the factorial procedure. The recursive implementation has to save the argument and the return address across the recursive call causing a stack frame to be created. The tail-recursive version does neither need a stack frame nor loads and stores to save the registers.

The recursive implementation incurs a significant amount of overhead to create and deallocate the stack frame and to save and restore the arguments that need to be saved across the recursive procedure call. Observing the evolution of the stack during an execution of the recursive implementation, one can see that the first n calls only allocate frames and store the respective argument on the stack. implicitly, it builds a list of all numbers from 1 to n and ever the same return address. Only when returning from the recursive calls the computation is performed.



2.8.2 The Calling Convention

Calling conventions regulate who has to save register contents when. Registers that the callee may change at its own discretion are called **caller-save** because the caller is responsible to save their contents before calling a function. Registers that the caller can assume to be unchanged by a call are called **callee-save** because the callee has to guarantee that when the function returns, these registers have the same contents as when the function was entered.

For efficiency reasons, callers pass the arguments to called functions in registers. In MIPS registers `$a0` to `$a3` are used for this purpose. The result of a function call (i.e. a value that the called function wants to return to the caller) is passed in `$v0`. The following table shows all MIPS registers and their role in the calling convention.

Table 2.8.3 MIPS registers with their short name, role in the calling convention, and purpose.

Number	Name	save	Comment
0	`$zero`	--	Always reads zero
1	`$at`	--	Reserved. Used by linker.
2 –3	`$v0–$v1`	caller	Return values
4 –7	`$a0–$a3`	caller	First four arguments
8 –15	`$t0–$t7`	caller	Temporaries
16–23	`$s0–$s7`	callee	Temporaries
24–25	`$t8–$t9`	caller	Temporaries
26–27	`$k0–$k1`	--	Reserved. Used by operating system
28	`$gp`	--	Global data pointer. Used to address elements in the global data segment.
29	`$sp`	callee	Stack pointer
30	`$fp`	callee	Frame pointer
31	`$ra`	caller	Return address

Chapter 3

Some Simple Algorithms

In this chapter we discuss several simple algorithms to exercise the various concepts of (MIPS) assembly programming that we have discussed.

3.1 Number Conversion

The first example strengthens our skills in some low-level bit fiddling. We have already seen that we can use a simple sys call (see Section A.2) to print the contents of a register in decimal numbers. In this section, we want to write a procedure that prints the 32-bit contents of a register *hexadecimally*. To this end, we recall that one hexadecimal digit can represent exactly four bits. So a 32-bit register content can nicely be represented by 8 hexadecimal digits.

The idea for our program is to extract four-bit packets out of the register from most significant to least significant, one after another. For each "packet", we convert the four bits into the respective ASCII code[5] of the character of the hexadecimal digit and print it.

For example, the four bits 1011 represent the number 11 and have the hexadecimal digit b. The ASCII code for the letter "b" is 98.

A brief look into an ASCII table reveals that the ASCII codes of decimal digits start at 48 and lower-case letters start at 97.

Let us focus first on the code that extracts the four bits and converts them into the ASCII character of the corresponding hexadecimal digit.

```
loop:
        srlv  $a0 $t0 $t1
        andi  $a0 $a0 15
        sltiu $t3 $a0 10
        bnez  $t3 smaller10
        addiu $a0 $a0 39
 smaller10:
        addiu $a0 $a0 48
```

Open in Browser

We assume here that the word to print is in register `$t0`. The first instruction is a shift that shifts the word to print down by `$t1` bits. `$t1` is supposed to index the four bit packets in the word and will assume the values 28, 24, 20, ..., 0 during the course of execution. To this end we will setup a loop later. Now, the shifts brings the four bits to be printed "down" so that the least significant bit of the packet is at bit position 0 of `$a0`. Since there may be several set bits in `$t0` above the four bits we

[5] see `https://en.wikipedia.org/wiki/ASCII`

want to print, we need to clear all bits beyond bit 3 by and-ing with 15. Now, `$a0` contains the four bits to print with the LSB at position 0.

The remaining code checks, if the four bits represent a value smaller than 10 in which case we want to compute the ASCII code of the corresponding decimal digit. This is done by the `addiu` instruction labeled `smaller10`. The branch at line 13 branches there if the four bits are smaller 10. If this is not the case, the branch "falls through" and the `addiu` with 39 is also executed. In that case, `$a0` contains a value x between 10 and 15 (including) and $39 + 48 + x = 87 + x$ which gives the ASCII code for the lower-case hexadecimal letter representing the values between 10 and 15.

Finally, we put a loop around this piece of code to iterate through all packets in `$t0`. The following listing shows the final version of the code with some main routine to test it.

```
 1       .data
 2  msg:
 3          .asciiz "Enter␣a␣number:␣"
 4      .text
 5  print_hex:
 6          move  $t0 $a0
 7          li    $t1 28
 8          li    $v0 11
 9  loop:
10          srlv  $a0 $t0 $t1
11          andi  $a0 $a0 15
12          sltiu $t3 $a0 10
13          bnez  $t3 smaller10
14          addiu $a0 $a0 39
15  smaller10:
16          addiu $a0 $a0 48
17          syscall
18          addiu $t1 $t1 -4
19  check:
20          bgez  $t1 loop
21          jr    $ra
22
23  main:
24          .globl main
25          la    $a0 msg
26          li    $v0 4
27          syscall
28          li    $v0 5
29          syscall
30          move  $a0 $v0
31          jal   print_hex
32          li    $v0, 10
33          syscall
```

Open in Browser

3.2 The Sieve of Eratosthenes

The sieve of Eratosthenes is an algorithm to enumerate all prime numbers smaller than a given number N. It uses a table with entries for each number from 2 to $N - 1$ and successively marks numbers in that table that are not prime. In the beginning, no table entry is marked.

To implement the table, we will use an **array**. An array is a data structure that stores a list of elements by putting them one after another in memory. Here, the list is the table of entries of 0 and 1 to indicate that a number is marked. Arrays have the appealing property that we can address an element in constant time by simply computing its address: If a is the address of element 0, we can simply compute the address element i by $a + i \cdot s$ where s is the size of an element in bytes (of course all elements have to be of the same "type" and therefore have to have the same size in order for this to work).

Note that we will use an array from 0 to $N - 1$ because it makes indexing simpler at the expense of two unused entries (0 and 1).

The idea of the algorithm is simple: We look at each number from 2 to $N - 1$ successively. If number q is not marked yet, we mark all its multiples $2q, 3q, \dots,$ smaller than N. For example, assume $N = 14$. When looking at table entry 3, we need to mark 6, 9, 12. All numbers not marked are prime.

Let's think about why this algorithm is correct. If we arrive at q we know, because we considered all numbers smaller than q before, that the multiples of each number $i < q$ have been marked. So if q is unmarked, it means that it does not divisible by any number bigger than 1. Therefore, q is prime.

The marking process can be sped up by the following observation: All multiples of q that contain a prime factor p less than q have already been marked when we considered p. So the smallest unmarked number is q^2 and it is sufficient to start marking there. Consequently, there is no sense in marking multiples of a number q' with $q' \cdot q' > N$.

Running time. Analyzing the running time of the optimized procedure is interesting. Let M be the greatest integer such that $M^2 < N$. For each $1 < i < M$ we are marking all multiples greater or equal to i^2 and smaller than N. So the overall number of markings performed is

$$\begin{aligned}\sum_{i=2}^{M} \lfloor (N - i^2)/i \rfloor &< \sum_{i=1}^{M} (N - i^2)/i \\ &= N \sum_{i=1}^{M} \frac{1}{i} - \sum_{i=1}^{M} i \\ &< N \ln M - \frac{M(M+1)}{2} < N \ln N\end{aligned}$$

In the last step, we have exploited that $1 + \ln n$ is an upper bound to the n-th harmonic number $1 + \frac{1}{2} + \cdots + \frac{1}{n}$.

Aside: This can be seen from $\int_1^x dz/z = \ln x$.

Let us finally give the implementation of the sieve of Erastothenes in MIPS assembly

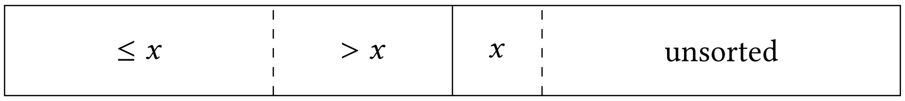
```
        .text
eratosthenes:
        .globl eratosthenes
        subu    $a2 $a1 $a0
        move    $t0 $a0

        # initialize tables with 0s
        b       zero_check
zero_loop:
        sb      $0 0($t0)
        addiu   $t0 $t0 1
```

```
zero_check:
        bne    $t0 $a1 zero_loop

        # we start with q=2
        li     $t0 2
era_loop:
        mul    $t9 $t0 $t0  # marking starts at q*q
        bge    $t9 $a2 end  # we don't need to mark
                            # if q*q > N
        addu   $t1 $a0 $t0  # compute address of entry q
        lb     $t1 ($t1)    # load entry
        bnez   $t1 no_prime # if marked, no prime, no
            marking

        # here, q (in $t0) is a prime
        # mark all multiples of q beginning from q*q
        addu   $t1 $a0 $t9  # compute address of entry of
            q*q
        li     $t8 1        # put a 1 into a register
        b      mark_check
mark_loop:                # loop to mark all multiples
        sb     $t8 ($t1)
        addu   $t1 $t1 $t0
mark_check:
        blt    $t1 $a1 mark_loop

no_prime:
        addiu  $t0 $t0 1    # set $t0 to q + 1
        b      era_loop
end:
        jr     $ra
```

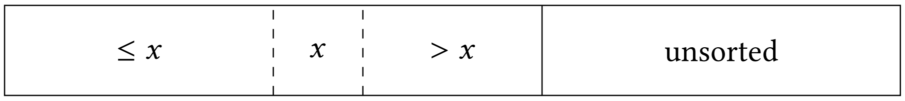
Open in Browser

3.3 Insertion Sort

Insertion sort is a simple sorting algorithm that mimics the way we sort small collections of things in our daily lives. Assume that we are given an array of n elements that are to be sorted. Insertion sort subdivides the array into two parts: A first part containing already sorted elements and the unsorted rest. In every step, insertion sort takes one element x from the unsorted part (it doesn't matter which one, typically one takes the first).



Then, it inserts at the right spot in the sorted part and moves the remaining sorted elements greater than the inserted element one up.



This step is repeated $n - 1$ times until the unsorted part is empty. Initially, the sorted part consists of just the first element of the array and the rest of the array constitutes the unsorted part.

Running time. When counting the number of comparisons, the worst-case running time of insertion sort is $O(n^2)$ because if the case that the input is already sorted, we need $i - 1$ comparisons when inserting the i-the element. This sums up to $1 + 2 + \cdots + n = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$ comparisons.

Finally, the following MIPS code implements insertion sort for integer arrays:

```
        .text
      # $a0 address of first element
      # $a1 address of element past the last
insertsort:
      addiu   $t0 $a0 4
outer_loop:
      beq     $t0 $a1 outer_end
      move    $t1 $t0
inner_loop:
      beq     $t1 $a0 inner_end
      lw      $t2 ($t1)
      lw      $t3 -4($t1)
      slt     $t4 $t3 $t2
      bnez    $t4 inner_end
      sw      $t2 -4($t1)
      sw      $t3 ($t1)
      addiu   $t1 $t1 -4
      b       inner_loop
inner_end:
      addiu   $t0 $t0 4
      b       outer_loop
outer_end:
      jr      $ra
```

Open in Browser

Exercises

1.

In practice, the number of memory movements also plays an important role in the run time of insertion sort. In which case do we have the worst-case number of memory operations? How does this relate to the number of comparisons?

3.4 Evaluating Polynomials with Horner's Method

Horner's method is a technique to evaluate a polynomial

$$p(x) = a_0 + a_1 x + \cdots + a_n x^n$$

for some value x. When implementing this formula in a computer program in the straightforward way, we need $1 + \cdots + n - 1$ multiplications for all the different powers of x. A more efficient imperative implementation could of course memorize the "last" power x^{i-1} of x and compute x^i by $x \cdot x^{i-1}$. This would then require two multiplications per summand: one to compute x^i and one for $a_i x^i$.

In contrast, Horner's method evaluates the polynomial with n multiplications. It relies on successively factoring out x in the term above, giving:

$$p(x) = a_0 + x(a_1 + \cdots + x(a_{n-2} + x(a_{n-1} + a_n x) \cdots))$$

This becomes more palpable when written down using tail-recursive [6] equations:

$$p'(x, 0, r) = r \cdot x + a_0$$
$$p'(x, i, r) = p'(x, i - 1, r \cdot x + a_i) \text{ for } i > 0$$

with

$$p(x) = p'(x, n, 0)$$

The following MIPS program implements this tail-recursive scheme using a simple loop (and no function calls). The function `horner` accepts three parameters: A pointer to the begin and end of the coefficient array and x. Register `$v0` is provisioned to hold the accumulator r and is initialized to a_n, the first element in the coefficient array. In each iteration of the loop, the accumulator is multiplied by x and added to a_i.

```
        .text
    .globl horner

        # $a0 Coefficients base
        # $a1 Coefficients end
        # $a2 x
horner:
        b       horner_head
horner_loop:
        lw      $t0 ($a0)
        mulu    $v0 $v0 $a2
        addu    $v0 $v0 $t0
        addiu   $a0 $a0 4
horner_head:
        bne     $a0 $a1 horner_loop
        jr      $ra
```

Open in Browser

3.5 The Tortoise and the Hare

In this section, we discuss an algorithm that checks if a linked list ends in a cycle. Linked lists are a another way of representing lists in a computer and differ from arrays in the following way: A linked list is a list in which each element contains the address of the next element of a list.

Aside: Because the elements of linked lists do not appear in memory contiguously, we cannot use address arithmetic to get to the i-the element in constant time. Instead we need to traverse the linked list from its root to the desired element.

If a linked list data structure is set up incorrectly, it may exhibit a cycle, i.e. a later element points to an earlier element in the list. In that case, the list is no list anymore but looks more like a “lasso”.

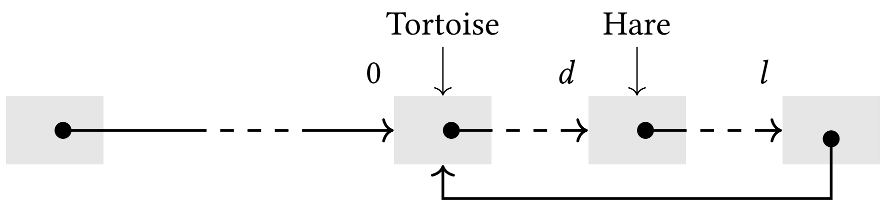
A straightforward algorithm to check if a linked list ends in a lasso is to traverse the list and keep a set of elements on the side to record the elements that have already been visited. If an element is visited that is already contained in the set, we have a cycle. However, this technique needs an additional data structure to implement this set (another list, for example). Furthermore, its runtime depends on the insert and lookup operations of the set.

The tortoise and the hare technique achieves the same goal without an additional data structure but just two pointers: The tortoise and the hare. The technique simply

[6]tail recursive means that the recursive invocation of a function is the result of the function and is not used in further calculations.

traverses the list. In each step, the tortoise advances one list element and the hare advances two elements. Now, the claim is that the tortoise and the hare meet (both pointers are equal) if and only if the list ends in a cycle. It is straightforward that they never meet if there is no cycle. But why do they meet if there is a cycle?

Assume that the list has n elements and ends in a cycle of l nodes. So, after $n - l$ steps, the tortoise reaches element 0 of the cycle. This element is the only element to which a pointer from outside the cycle points to. Assume that the hare is at element d of the cycle. Now, in each step the distance from hare to tortoise will *decrease* by one. (The hare is in fact moving one element away from the tortoise in each step, but because they run on the cycle, it is actually moving closer.) So, after $l - d$ steps they stand on the same element.



The following MIPS code implements this algorithm. It assumes that each list element contains some payload data of four bytes (maybe a word) and a pointer to the next element at offset 4.

```
        .text
        .globl tortoise_and_hare

        # $a0: address of first element
tortoise_and_hare:
        # $a0 Hase
        # $a1 Igel
        move    $a1 $a0
loop:
        beqz    $a0 not_cyclic
        lw      $a0 4($a0)
        beqz    $a0 not_cyclic
        lw      $a0 4($a0)
        lw      $a1 4($a1)
        beq     $a0 $a1 cyclic
        b       loop
cyclic:
        li      $v0 1
        jr      $ra
not_cyclic:
        li      $v0 0
        jr      $ra
```

Open in Browser

Chapter 4

Introduction to C

Programming with assembly languages is cumbersome, as we have seen in the previous chapter. For this reason, software developers write their code in high-level programming languages. A **compiler** then translates the program of the high-level language to the machine language. In this process, it handles many of the inevitable troubles of programming in assembly languages:

- Assembly programs are not portable. They are always written in the instruction set of a processor family. To run the program on a different processor, it needs to be rewritten entirely. A program written in a high-level language, if there is a compiler for the new processor, can just be re-compiled.
- Memory cells in assembly programming are unnamed and can only be accessed by number. The resulting programs are hard to read and understand for programmers. Furthermore, we need to decide which values are held in registers and when they are written to memory. This is particularly problematic if there are not enough registers available and when calling conventions need to be implemented.
- Assembly languages provide little syntactical structure. Convenient concepts like the infix notation of mathematical expressions cannot be used in assembly languages. Complex expressions need to be serialized by hand and the intermediate results need to be stored to registers.
- Control structures like "Do … until a condition is satisfied" or "if a condition is fulfilled, do *A*, otherwise do *B*" are cumbersome to implement with branching instructions.

In this chapter, we study the programming language C,which remedies all of the above points. C provides abstractions to the programmer that hide the specifics of the underlying machines. As C programmers expect a translation to efficient machine code, C only provides as much abstraction as necessary to write portable programs without causing overhead. For this reason, some people refer to C as a "high-level assembler". This intention gives C problematic properties that we will discuss later in this chapter (Section 4.15). These properties cause many severe security vulnerabilities in software written in C.

C is a very successful programming language. Every modern operating system is, because of C's close relation to the machine language, written in C. Software for which memory consumption and execution time are important are written in C. C does not need a complex run-time environment and is rather easy to translate into machine language. This renders C the default language when developing embedded systems. Moreover, C had a strong influence on many other successful languages like C++, C#, Java, JavaScript, etc.).

4.1 Basics

```
 1  int factorial(int n) {
 2    int res;           /* declaration of a variable */
 3    res = 1;           /* statement: assignment expression */
 4    while (n > 1) {    /* statement: loop */
 5      res = res * n;   /* statement: assignment expression */
 6      n   = n - 1;     /* statement: assignment expression */
 7    }
 8    return res;        /* statement: return to caller */
 9  }
10
11  int main() {
12    int res;
13    res = factorial(3);
14    printf("%d\n", res);
15    return 0;
16  }
```

Open in Browser

Listing 4.1.1 A C program consisting of two functions. It computes the factorial of 3 and writes it on the screen.

4.1.1 Static Properties

Consider a first C program in Listing 4.1.1. We start by discussing its syntactical properties and the properties of the static semantics. These are properties that the program has, independently of any concrete input. They therefore do not concern individual program executions, but the structure of the program.

Every C program consists of at least one file. Each such file constitutes a **translation unit**.

In every translation unit, there is a list of **declarations**. Every declaration establishes a name, or **identifier**, for some entity. The letter sequences `factorial`, `n`, `res`, `main`, and `printf` in Listing 4.1.1 are identifiers. The other, highlighted words are **keywords**. They are markers for specific syntactic constructs (e.g., `while`) or are predefined identifiers (e.g., `int`). Identifiers can occur in two ways: *defining* and *using*. Our example shows a translation unit where two **functions**, `factorial` and `main`, are declared. The occurrence of `factorial` in line 1 is defining, as the identifier `factorial` is established for the function there, whereas the occurrence in line 12 is using, since `factorial` here refers to the previously defined function.

Remark 4.1.2 Although the C language specification calls these constructs “functions”, they are not functions in the mathematical sense. Besides computing values, C’s functions can affect memory and interact with the user. Executing a function twice with the same arguments can therefore easily result in different values. C’s functions correspond to the subroutines that we have seen in Section 2.8

Consider first the function, `factorial`. The function declaration in line 1 establishes that `factorial` has a **parameter** `n` of the type `int` and the return type `int`. The function’s code, its **body**, consists of a **block**: A sequence of **statements** enclosed in braces. Each statement that does not end itself with a nested block is terminated by a semicolon. Therefore, the outer block of our example program consists of four statements in the lines 2, 3, 4 and 8. The while-loop in line 4 is a statement that contains another, nested block with two more statements.

Aside: Statements, which are also referred to as "commands", are characteristic for imperative programming languages, hence the name that is derived from Latin *imperare* = (to) command.

We separate statements into four categories: variable declarations, expressions, blocks, and control-flow constructs. We will discuss them in more detail in the following sections.

The first statement in `factorial`'s body is a **variable declaration**. These always consist of a **type**, in this case `int`, and an identifier, here: `res`. The identifier that occurs in a variable declaration is called a **variable**. The occurrence of a variable in a declaration is defining. All other occurrences are using. The parameter `n` in line 1 is a variable as well.

Remark 4.1.3 The term "variable" is not used consistently in literature. We follow the usual naming from mathematics, where the occurrence of a letter (sequence) in a term is called a variable. Nevertheless, the *meaning* of variables in imperative programs differs from mathematics, see Remark 4.3.3.

Every declaration has a **scope**. The scope of a declaration for an identifier determines the part of the program text in which another (using) occurrence of the identifier refers to this declaration. For example, the variable `res` is in scope (or *visible*) in the entire body of `factorial` (starting from its declaration). Therefore, the occurrences in the statement `res = res * n` refer to this declaration. The scope of a *local variable*, i.e., one defined inside a function, starts at its declaration and ends with the innermost block that encloses its declaration.

Remark 4.1.4 Since blocks can be nested, variable declarations can be shadowed, as in this example:

```
1  void foo(int x) {
2      int y = 0;
3      if (x > 0) {
4          int y = 0;
5          y += 1;
6      }
7  }
```

Open in Browser

Here, the using occurrence of `y` inside the `if` statement refers to the innermost declaration, not the one in the second line.

4.1.2 Dynamic Properties

We now informally discuss the dynamic properties, i.e. the semantics of C. These are properties that determine what happens when we execute a C Program. The execution model of imperative languages (and particularly C) is very closely derived from the behavior of the underlying machines. As discussed in the introduction to this chapter, imperative languages provide abstractions for the fundamental building blocks of a machine language, to enable programming "close to the hardware."

(Imperative) variables, whose static properties we have briefly discussed in the previous section, are an abstraction that is characteristic of imperative programming. The dynamic behavior of a variable in imperative programming differs from the similarly-named concepts in mathematics (and *functional* programming). As this behavior is central for understanding imperative programming languages, we discuss it in Section 4.3 in more detail. For now, it is sufficient to understand that each variable during execution denotes a container that can carry any value of some type. For example, executing the variable declaration in line 2 allocates a new container that can carry a value of type `int`.

The execution of every C program starts with the function declared with the name `main`. For our example, this is line 12. Just like the instructions of an assembly program, the statements in a function's body are executed one after the other. The first statement (in line 12) allocates a new container to carry an `int` value and binds it to the identifier `res`. Afterwards, the function `factorial` is called (line 13). When calling a function, new containers for all parameters of the called function are allocated. In the example, this is one container, for the parameter `n`. Then, the expressions in the parentheses after the name of the called function (separated by comma; in the example, there is only one: `3`) are evaluated. The resulting values are the **arguments** or **actual parameters**. These values are placed into the containers for the corresponding parameters. In our example, the `3` is placed into the container bound to `n`. The program execution then continues in the called function, `factorial`.

The statements of a block are executed in sequence. The function's execution therefore starts in line 2 by allocating a container for `factorial`'s `res` variable. The statement in line 3 evaluates the expression on the right-hand side of the assignment operators (the constant `1`) and writes the resulting value to the memory location given by the left-hand side (the previously allocated container for `res`).

The `while` loop evaluates its **break condition** (here `n > 1`). If the result is not equal to 0, it executes the statement of its body (here a nested block). These two steps are repeated until the break condition evaluates to 0. Within the loop body, there are two assignments that are evaluated as described for line 3.

After termination of the loop, the `return` statement returns the execution to the calling function, `main`, providing the current value of `factorial`'s variable `res` as the return value. In `main`, the assignment in line 13 is completed by writing the return value to the variable `res`. Next, the `printf` function is called with two arguments, which prints the value of `res` to the screen. Afterwards, the program execution is terminated with the execution of the `return` statement in the last line.

4.2 Execution Traces

When programming, it is helpful to grasp the meaning of a program part by executing the code "in ones head". For this purpose, we need to precisely understand the meaning of each statement of the program. This understanding is best practiced by writing **execution traces** of small programs on paper.

Aside: We will formally discuss the semantics of a subset of C, and in particular the notion of a program state, in Chapter 6. Here, we content ourselves with an informal depiction.

line	res	side effect
11		
12		
13	?	call to `factorial` →
14	6	print `res`
15		termination

(a) Call to `main`

line	`n`	`res`	side effect
1	3		
2			
3		?	
4		1	
5			
6		3	
7	2		
4			
5			
6		6	
7	1		
4			
8			return with the value 6

(b) Call to `factorial`

Figure 4.2.1 Execution trace of the program in Listing 4.1.1.

We represent execution traces in table form. Each line of a table represents the execution of a statement and visualizes the values of the variables *before executing* the statement. In the first column, we note the line number of the executed statement. For each variable declaration, we add another column with the declaration's identifier as its title. For simplicity, we assume that no declaration is shadowed by another one in a nested scope. A statement can have different *effects* on the **state** of the execution: For example, a variable declaration can allocate new containers, the end of a block can deallocate existing containers, and an assignment may read from or write to containers. In a line where the previous statement did not change a variable's content, we leave its column clear. The content of a freshly allocated container is undefined, which we represent with a question mark. Besides changes to contents of a variable's container, a statement can have "side effects", for example calling a function or printing text on the screen. We note these in a column with the title "side effects".

Aside: Of course, a line of a program may contain more than one statement. For simplicity, we assume that this is not the case.

Every call of a function requires a new table in the execution trace. In our example, that is the call to `main` that starts the program execution and the contained call to `factorial`. Execution traces can therefore be nested. We mark the connection to the called function's table in the "side effects" column of the calling function. As functions can be called multiple times and can even call themselves (directly or indirectly), there can be multiple tables for the same function in an execution trace. It is important to note that each call of a function has a separate binding to containers for its variables.

4.3 Imperative Variables

Computations are done with **values**. For example, 2 is a value. A **type** is a set of values. Aside from the `void` type, which represents the empty set of values, we distinguish scalar and aggregate types. The values of scalar types, in contrast to the values of aggregate types, are not composed of other values. For example, 2 is a value of the scalar type `int` and `{3, 4}` is a value of the aggregate type `struct {int a, b;}`, which represents pairs of `int`s.

During the program's running time, values are stored in **containers**.

Remark 4.3.1 The C language specification [1] (or C Standard) calls these “object”. Since this term is very generic and can easily lead to confusions, we use the term “container”.

Every container has a *size* (in bytes), an *address* and a *life time*. The address uniquely identifies a container at every point in time. There are no two different containers with the same address.

The addresses of containers are also values. Consequently, there may be containers whose values are addresses of other containers. Such containers are called *pointers*. We discuss them in more detail in Section 4.10.

Variables are identifiers that occur in a variable declaration. Such a variable declaration consists of the variable itself and a type. For example, `int x;` declares a variable `x` of type `int`. When executing a variable declaration, two actions are performed: First, a new container is allocated with a size derived from the type of the variable declaration. If, for example, an `int` value has a size of 4 bytes, the container bound to `x` needs to have a size of at least 4 bytes. Second, the variable is bound to the container’s address, it *references* the container. This link persists until the end of the variables scope, where the container is deallocated.

Aside: In C, every type `T` other than the empty type `void` has the size `sizeof(T)`.

Remark 4.3.2 Nomenclature. In practice, the term “variable” is commonly used for both: The identifier and the container. When talking about a concrete execution, for example, one can say “the variable `x` has the value 5” and mean that the container that is bound to the variable `x` contains the value 5.

Remark 4.3.3 Imperative variables compared to mathematical variables. In functional programming languages as well as in mathematics, variables have a different meaning than in imperative programming languages. Essentially, both paradigms, functional and imperative, agree that variables are immutably bound when they are declared. In functional languages, they are bound to a value, whereas, in imperative programming, the binding is to the address of a container whose content is mutable. Consequently, there are two ways to evaluate a variable in an imperative programming language: To the address of its bound container (L-evaluation) and to its contents (R-evaluation). We discuss these in detail in Subsection 4.8.2 and Chapter 6.

A container’s life time determines when it is allocated and deallocated during program execution. There are several kinds of life times:

4.3.1 Local Variables

Containers for local variables are allocated and bound to their identifier when executing a variable declaration. They are deallocated after the last statement of the innermost block that includes the variable declaration. Therefore, we can refer to a local variable anywhere after its declaration within the innermost surrounding block that surrounds its declaration.

Reconsider the example in Remark 4.1.4. When the declaration in line 2 is executed, a container large enough for an `int` is being allocated for local variable `y` and the value 0 is stored in it. Assume the identifier `x` has a value greater than 0 so that the if condition in line 3 is true. When line 4 is executed, a new container for an `int` is allocated and bound to `y`. Note that the container allocated by line 2 is now no longer accessible through the identifier `y`: The inner declaration of `y` shadows the outer one. After the then-block has finished executing in line 5, the container for `y` is deallocated and the old binding for `y` is being restored. When block is left, the container allocated in line 2 will again be accessible through the indentifier `y`.

4.3.2 Global Variables

Global Variables are declared outside of function bodies. Containers for them are sized an bound to their identifiers just as for local variables. The difference is that a global variable's life time spans from program start until program end. It is therefore available at all times, in all subsequently declared functions.

Aside: Global variables can also be declared within a function body, when using the storage-class specifiers `static` or `extern`. We will not treat these here.

Use of global variables is a bad practice and should therefore, whenever possible, be avoided. The motivation for this guide line is that global variables make it very easy to introduce hard-to-retrace dependencies between different functions. When we use global variables, we can therefore no longer understand and verify a function individually outside of its calling context.

Example 4.3.4 Global Variable. Consider the following function:

```c
int data[1024];

void sort() {
    /* sort the values in data */
}
```

Open in Browser

This sorting function can only operate if the data to be sorted has been copied to the array `data` before. This reduces its flexibility and makes the function difficult to comprehend: Its parameters do not say what is to be sorted. A better design would give `sort` a parameter with a pointer type that should point to the array that needs to be sorted and a parameter for the length of the array. □

4.3.3 Dynamically Allocated Containers

Commonly, the number and size of the required containers in a program depends on its input and is not known statically. We need to allocate such variables dynamically. The C function `malloc` accepts a number of bytes as an argument, allocates a new container of this size, and returns its address. The container can be deallocated with a call to `free` with its address as the argument.

Aside: "Statically" means: Without executing the program with specific inputs.

Containers that are not eventually deallocated are a programming error and cause *memory leaks*. Especially for longer-running programs, these are a serious problem as they can progressively leak all available memory of the system.

Example 4.3.5 Dynamically Allocated Containers. Consider the following function:

```c
int *unit_vector(int dimension, int n_dimensions) {
  int *res = malloc(sizeof(res[0]) * n_dimensions);
  for (int i = 0; i < n_dimensions; i++)
      res[i] = 0;
  res[dimension] = 1;
  return res;
}
```

Open in Browser

The statement in the first line allocates two new containers: The first one can carry an *address*; its address is bound to the variable `res`. The second one originates from the call to `malloc`. The argument to the `malloc` call determines the size of this container. `sizeof(res[0])` is the size of an `int` (see Section 4.10), and the container should have `n_dimensions`-times this size. So, this call to `malloc` requests a container that is large enough to carry `n_dimensions` many `int`s. `malloc` returns the address of such a container, which is then stored in the container bound to `res`.

The subsequent loop writes into each space for an `int` in the container a 0. In the following statement, a 1 is written to the `dimension`-th position of the container. Lastly, the function returns the address of the allocated container to its caller.

It is noteworthy that the life time of the dynamically (with `malloc`) allocated container does not end with the return to the calling function, in contrast to the local variable `res` that carries its address. □

We will discuss dynamic memory allocation in more detail in Section 4.12.

4.4 The C Memory Model

We have already seen that containers have addresses. Many programmers believe that C's notion of an address is identical to address notion of a processor that we have seen in Chapter 2. This is however not true. In particular, C does not require the address of a container to be an address in the main memory of the computer. Even more, while C allows some address arithmetic (computing new addresses by adding or subtracting offsets to the **base address** of the container), this address arithmetic is restricted to the container the base address belongs to. It is in fact **undefined behavior** (we will talk about that later in Section 4.15. For now just think of undefined behavior as something equally bad as dividing by 0) to create an **out-of-bounds** address by address arithmetic. These restrictions may seem counterintuitive and limiting at first, since C is intended to enable programming "close to the hardware". However, they allow compilers (the tools that translate C programs to machine code) more freedom when deciding where containers are actually located and make the meaning of programs independent of where containers actually reside. Let us consider the following code snippet to better understand why C makes these restrictions.

```c
int foo() {
  int a = 1;
  int b = 2;
  int* p = &b;
  p = p + 4;
  *p = 42;
  printf("%d\n", a);
}
```

Open in Browser

The function `foo` defines two local variables `a` and `b` and initializes them to 1 and 2. The third local variable `p` is a **pointer**, i.e. a variable that can hold an address. It is initialized to the address of variable `b`. In the next statement the programmer adds 4 to the address of `b`. Maybe the programmer believed that all local variables reside in a stack frame (see Section 2.8) in the order they were declared in the program. And maybe he thought that an int is four bytes long, so `p + 4` would essentially give him the address of `a`. But this is far from true. This program actually has *undefined behavior* because the address `p + 4` is invalid. It is out of the bounds of `b`'s container.

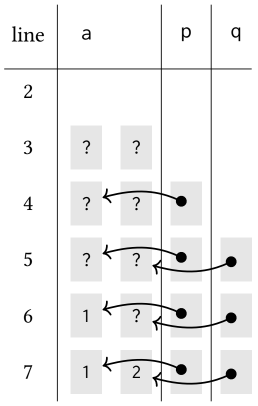
Let us briefly ponder on what would happen if the program above actually had the semantics (meaning) our inexperienced programmer had in mind. Then, the program's behavior actually was defined and it printed 42. However, this would also entail that if swapped the order in which `a` and `b` were declared the program would do something else! We certainly do not want to make our program's behavior dependent on the order in which variables are declared. Furthermore, giving our example program a meaning would also entail that the compiler is much more limited in where it allocates the containers' memory. If address arithmetic on a base address could yield a valid address of another container, it would be much harder for a compiler to allocate a container to a processor register (which is certainly much more pleasant because we have seen that registers are much faster operate on than memory). The more abstract memory model C defines permits the compiler to hold the value of a local variable in a register as long as the address of that variable has not been taken.

As mentioned before, C does however allow containers for compound data that have space for more than one value. The separate components of such a container can be addressed individually. We can obtain the address of a byte within a container by adding an offset to the containers address. There can therefore be multiple different addresses to the same container that refer to different positions in the container. Formally, an address is a pair of the container address and an offset, which needs to be a number between 0 and the container's size. Addresses that refer to the same container are totally ordered. This enables pointer arithmetic, which we discuss in Section 4.10.

Example 4.4.1 Consider the following function and an execution trace for it:

```
1 void foo() {
2   int a[2];
3   int *p = a;
4   int *q = a + 1;
5   *p = 1;
6   *q = 2;
7 }
```

Open in Browser



Line 2 allocates a container for two `int`s as a local variable `a`. Lines 3 and 4 allocate more containers, bound to `p` and `q`, both of which can carry addresses. In line 3, the address of `a`'s first `int` is stored in `p`'s container. Similarly, in line 4, the address of `a`'s second `int` is stored in `q`'s container. The last two statements store the values 1 and 2 in the first and second `int` component of `a`. In execution traces, we denote addresses as arrows to the referenced container. □

4.5 Translation Units

A C program usually consists of multiple translation units. A translation unit is a file with the file ending `.c`. The C compiler translates the functions of a translation unit into object code, which it stores in a binary file with the file ending `.o`. In a Unix system, the command

```
$ cc -o x.o -c x.c
```

translates the translation unit `x.c` into the binary file `x.o`. The C compiler can than link multiple such translated translation units to an executable file:

```
$ cc -o prg x.o y.o z.o
```

Here, `prg` is the name of the to be produced executable program and can be chosen freely. Afterwards, the current directory contains a file `prg` that can be started with

```
$ ./prg
```

Aside: The $ marks the command prompt of the Unix shell.

Remark 4.5.1 Separating the translation into these steps is helpful for larger projects for which the translation can take a long time: If only one translation unit is changed after a previous build, the compiler only needs to re-compile the changed translation unit and perform the final linking step. Additionally, separate translation units can be translated to object code independently, which makes this step easy to parallelize. This can speed up large builds considerably on modern systems.

The C compiler can find errors in the program code and issue warnings in both stages of the translation process. While the warnings do not abort the translation process, they should be considered carefully as they can hint towards subtle programming errors. A good C program should be translated by the compiler without warnings.

4.5.1 main

To successfully build an executable, exactly one contributing translation unit needs to contain a function with the name `main`. Program execution starts with this function. Unix programs (and, consequently, C programs) can be started with arguments. These are given to the `main` function in the form of two parameters, `argc` and `argv`. `argc` contains the number of the provided program arguments, including the program name as mandatory first argument. The actual character strings of the arguments are available in the `argv` array. A character string, usually just called *string* in C is a null-terminated sequence of characters. Strings are referred to by the address of their first character. `argv` is therefore an array of pointers to the first character of each argument (see Figure 4.5.2).

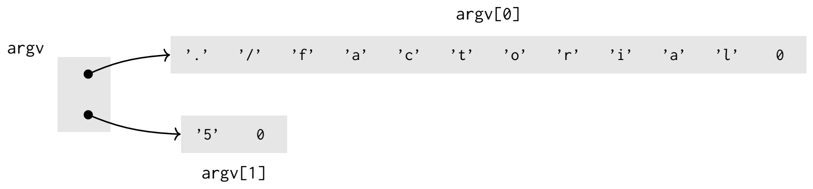


Figure 4.5.2 Depiction of the variable `argv` when starting the program as `./factorial 5`.

For an example, consider a main function that calls the `factorial` function from the previous section with an argument obtained from the command line.

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

/* here is the declaration of the factorial function  */

int main(int argc, char *argv[]) {
    if (argc <= 1) {
        fprintf(stderr, "syntax:␣%s␣<value>\n",
                argv[0]);
        return 1;
    }

    int n = atoi(argv[1]);
    int r = factorial(n);
    printf("%u\n", r);
    return 0;
}
```

Open in Browser

Listing 4.5.3 A `main` function for the factorial function.

We build this program with the following commands:

```
$ cc -o factorial.o -c factorial.c
$ cc -o factorial factorial.o
```

Then, the following program execution will fail:

```
$ ./factorial
syntax: ./factorial <value>
```

The provided message tells us to provide a number whose factorial the program should compute. The following invocation will produce the desired result:

```
$ ./factorial 5
120
```

The `main` function first checks whether an argument was provided. This is the case if the value of `argc` is greater than 1 (since the program name is always the first parameter.) If no argument was given, the program prints an explanatory message to the user and terminates with the value 1. Otherwise, the first argument (a string of characters) is converted to an integer number and its factorial is computed. The program displays the result via `printf` and then terminates successfully with the value 0.

Remark 4.5.4 In Unix, every program can provide an "exit code" upon termination. In a C program, this is the return value of the `main` function. By convention, an exit code of 0 signifies a successful execution, whereas other numbers can encode different errors.

4.5.2 Calling Functions from Other Translation Units

Let us assume that we want to separate the `main` function and the `factorial` function into different translation units. It is often good practice to bundle functions that are thematically connected, for example because they operate on similar data, into their own translation unit. We usually separate the `main` function from other functions since it contains mostly argument handling and gives the relevant arguments to the other functions. The `factorial` function could be reused in a different project where factorials need to be computed; our `main` function less so. Therefore, it is reasonable to separate the functions into two translation units, which are compiled separately.

```c
#include "factorial.h"

int factorial(int n) {
    int res = 1;
    while (n > 1) {
        res = res * n;
        n   = n - 1;
    }
    return res;
}
```

Open in Browser

(a) `factorial.c`

```c
#ifndef FACTORIAL_H
#define FACTORIAL_H
int factorial(int n);
#endif /* FACTORIAL_H */
```

Open in Browser

(b) `factorial.h`

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

#include "factorial.h"

int main(int argc, char *argv[]) {
    if (argc <= 1) {
        fprintf(stderr,
            "syntax:␣%s␣<value>\n",
            argv[0]);
        return 1;
    }

    int n = atoi(argv[1]);
    int r = factorial(n);
    printf("%u\n", r);
    return 0;
}
```

Open in Browser

(c) `main.c`

Figure 4.5.5 Factorial function and `main` function in two separate translation units. The header file `factorial.h` contains the prototype of the function `factorial`. The preprocessing directives (`#ifdef`, etc.) ensure that the file content is only included once per translation unit. While not necessary here, this convention becomes important when header files include other header files, to break infinite recursive include sequences.

When we delete the `factorial` function from the translation unit in Listing 4.5.3, the compiler rejects the translation unit since it does not know `factorial`'s type. For a successful translation, the compiler needs to know the type of every called function. [7] The type of `factorial` can be established in the `main.c` translation unit by providing the **prototype** of `factorial`. The prototype of a function consists of its name, its return type, and the types of its parameters:

[7]The parameter types contained in the function's type tell the compiler which parameters are to be passed in which register (Subsection 2.8.2). All translation units that declare or call a function need to agree on the same prototype to ensure that the code generated for the call sites interacts correctly with the function.

```
int factorial(int);
```

This is commonly called a **function declaration**, in contrast to a **function definition** where additionally, the function's code is provided in a body. In practice, we do not manually duplicate the prototype of every function into every translation unit in which it is used. On the one hand, this would require writing a lot of redundant code. On the other hand, it would be prone to errors since all translation units would need to be adjusted if we change the function, e.g., by adding or changing a parameter. For this reason, we create **header files** that are included by the **C preprocessor**.

Aside: In a project, there can be declarations for a function in many translation unit, but there may only be one definition.

The C preprocessor is a separate program that is invoked by the C compiler before it performs the actual translation. It transforms a text into a new text by expanding preprocessing directives. All preprocessing directives start with a hash sign (#). The directive `#include "x.h"` for example interrupts the preprocessing of the current file, preprocesses the file `x.h`, and then resumes preprocessing the original file. As a result, in our example in Figure 4.5.5, the preprocessing inserts the content of `factorial.h` into both translation units, `factorial.c` and `main.c`.

Aside: The text does not need to be a C program. Some infrastructures for other programming languages use the C preprocessor as well.

4.5.3 Makefiles

In practice, projects can easily consist of hundreds to thousands of translation units. To avoid building them all by hand, there is the Unix tool `make`. It operates based on a file with the name `Makefile`, in which we specify how to build the project. This description contains the *dependencies* between the involved files and a description how to produce files from their prerequisites.

Aside: `make` also keeps track of the files that need not be recompiled for the linking step: If the prerequisites of a target have not been modified since the target file was last built, it is not rebuilt.

```
factorial: factorial.o main.o
        cc -o $@ $^

main.o: main.c factorial.h
factorial.o: factorial.c factorial.h

%.o: %.c
        cc -o $@ -c $<

clean:
        rm -f factorial *.o
```

Listing 4.5.6 A simple `Makefile` for the factorial program.

The first two lines of the `Makefile` specify that we need the files `factorial.o` and `main.o` to build the file `factorial`, and that the latter file is built from the former two files with the command `cc -o factorial factorial.o main.o`. The last two lines determine that any file ending with `.o` is built from a similarly named file with ending with `.c`. The command `cc -o x.o -c x.c` performs this translation.

The placeholders in the build rules have the following meaning:

- `$@`: the "target" of the rule, i.e., the text on the left of the colon

- `$<`: the first “prerequisite” of the target, i.e., the first word on right of the colon
- `$^`: all prerequisites of the target, i.e. all words on right of the colon

Usage of `make` is not restricted to C. It is a general tool to describe build processes and dependencies. It is however most commonly used for C projects.

4.6 Number Types

Every variable in C has a type. The type of a variable determines what values can be stored in the container bound to the variable. We distinguish **scalar types** and **aggregate types**. The scalar types in C include different kinds of number representations with different ranges. There are integer types and floating-point types. The integer types `char`, `short`, `int`, and `long` come in two flavors, `unsigned` and `signed`, which are prepended to the type. If no signedness is specified, the type is implicitly `signed`. If only `unsigned`, without any further type specifier, is used, it refers to an `unsigned int`.

Aside: An exception to this rule is `char`: According to the C language specification the default signedness of this type can differ for different compilers. Use `signed char` or `unsigned char` if signedness is relevant.

For `signed` integer types, the C Standard [1] does not define which representation is used. In particular, one should not assume that `signed` values are represented in two’s complement and the corresponding arithmetic is used. The `unsigned` integer types are defined to use modulo arithmetic: They wrap around upon overflow. The Standard also does not specify the sizes of the integer types. It only guarantees the following relationships:

$$1 = \texttt{sizeof(char)} \leq \texttt{sizeof(short)} \leq \texttt{sizeof(int)} \leq \texttt{sizeof(long)}$$

The `sizeof` operator in C provides the size of any type. This weak specification is intentional, as it enables correct C implementations on a variety of different computers. It is however also the cause of many software bugs, since it is easy for uninformed programmers to make too strong assumptions about the semantics of C. For ordinary calculations, we use `int`, since an `int` usually corresponds to a machine word.

Floating-point types are used in numerical programs. They represent an excerpt of the rational numbers. As there are is only a finite number of bits available to represent a value, computations with floating-point values generally involve rounding errors. In longer-running programs, these can propagate without notice, leading to program errors that are hard to detect. It is tempting to program with floating-point values as if they were rational or even real numbers. However, floating-point arithmetic violates many of the essential properties of real and rational arithmetic that we are used to rely on, such as the associative law. C provides three floating-point types: `float`, `double`, `long double`. They differ in their size (and therefore in their range and precision).

4.7 Control Flow

C provides several language constructs to control the program flow. These determine which statements are executed and when they are executed.

4.7.1 Selection Statements

The if-statement `if (C) A else B` evaluates the condition `C`. If its value is unequal to 0, statement `A` is executed, otherwise `B`:

Aside: The `else` branch can be omitted if it is not required.

```c
void print_max(int a, int b) {
    if (a < b)
        printf("%d\n", b);
    else
        printf("%d\n", a);
}
```

Open in Browser

Listing 4.7.1 Example program using an if-statement

Another selection statement is the switch-statement. It evaluates an expression of type `int` and jumps to different `case` labels depending on the result:

Aside: Enumeration types in C, which are declared with the `enum` keyword, introduce names for integer values that can be used to make code easier to understand.

```c
enum { jan, feb, mar, apr, may, jun, jul, aug, sep, oct,
    nov, dec };

int days_per_month(unsigned month, bool is_leap_year) {
    assert (month < 12);
    switch (month) {
    case jan:
    case mar:
    case may:
    case jul:
    case aug:
    case oct:
    case dec:
        return 31;
    case feb:
        return 28 + is_leap_year;
    default:
        return 30;
    }
}
```

Open in Browser

Listing 4.7.2 Example program using a switch-statement with an enumeration type. The `assert` statement in the first line of the function body validates when executed that the function is called with a number corresponding to a valid month and aborts the execution if that is not the case.

If the selecting value fits to no provided `case` label, execution jumps to the `default` label. It is important to note that after execution jumps to a label, *all* statements until the end of the `switch` block are executed. If this behavior is undesired, the `switch` block can be left earlier with a `break;` or a `return` statement.

4.7.2 Loops

C has three kinds of loops. The do-while-loop `do A while (C)` executes the statement `A` and then checks the condition `C` to determine if the loop needs to be executed again:

```
int sum(void) {
    int sum = 0;
    int n;
    do {
        printf("Enter␣a␣number␣(0␣=␣finish):␣");
        scanf("%d", &n);
        sum += n;
    } while (n != 0);
    printf("The␣sum␣is␣%d\n", sum);
}
```

Open in Browser

Listing 4.7.3 Example program using a do-while-loop

Characteristically, the loop body `A` is always executed at least once, independently of `C`.

The while-loop `while (C) A` repeats `A` as long as the condition `c` is non-zero when it is checked before an iteration. In contrast to the do-while-loop, `A` is never executed if `C` is not satisfied before executing the loop.

```
int sum(int *a, int n) {
    int res = 0;
    int i = 0;
    while (i < n) {
        res += a[i];
        i = i + 1;
    }
    return res;
}
```

Open in Browser

Listing 4.7.4 Example program using a while-loop

The for-loop is an abbreviation of a while-loop to compactly declare iteration variables:

```
int sum(int *a, int n) {
    int res = 0;
    for (int i = 0; i < n; i++)
        res += a[i];
    return res;
}
```

Open in Browser

Listing 4.7.5 Example program using a for-loop

The break-statement can be used to leave the innermost surrounding loop:

```c
void sum(void) {
    int sum = 0;
    for (;;) {  // is equal to while (true)
        int n;
        printf("Enter␣a␣number␣(0␣=␣finish):␣");
        scanf("%d", &n);
        if (n == 0)
            break;
        sum += n;
    }
    printf("The␣sum␣is␣%d\n", sum);
}
```

Open in Browser

Listing 4.7.6 Example program using a break-statement. `scanf` is a function to query for inputs from the user.

The continue-statement jumps directly to the end of the loop body. For do-while and while-loops, this means that the termination condition is checked in the next step, in for-loops the increment portion of the loop is executed first.

```c
void cycle(int mod) {
    for (int i = 0; ; i++) {
        if (i == mod) {
            i = 0;
            continue;
        }
        printf("%d\n", i);
    }
}
```

Open in Browser

Listing 4.7.7 Example program using a continue-statement

4.7.3 Function Return

The return-statement, as we have seen before, ends the execution of the current function and returns the program flow to the calling function. Unless the function has a void return type, the return-statement requires a return value:

```c
int max(int a, int b) {
    if (a < b)
        return b;
    else
        return a;
}
```

Open in Browser

Listing 4.7.8 Example program using a return-statement

4.8 Expressions

In C, computations are represented as expressions. C provides a number of binary and unary arithmetic operators. Some of them are overloaded: The same symbol is used for different operand types. For example, the processor needs to perform different operations for adding two floating-point numbers than for adding two integers, but both are represented with the same symbol in C. The C compiler uses the types of the operands to determine which operation is used; the overloading is therefore resolved during the compilation.

Aside: In languages like JavaScript, where the types cannot be determined statically, the appropriate operations have to be selected during the program's execution, which leads to a slower execution speed.

For all types $i \in \{$`int`, `long`$\}$ and all numerical types $f \in \{$`int`, `long`, `float`, `double`$\}$, C provides the following arithmetic operators:

Table 4.8.1 Arithmetic operators in C

Symbol	Signature	Description
`~`	$i \longrightarrow i$	bitwise negation
`!`	$i \longrightarrow$ `int`	logical negation
`+ -`	$f \longrightarrow f$	unary plus, minus
`+ - * /`	$f \times f \longrightarrow f$	addition, subtraction, multiplication, division
`%`	$i \times i \longrightarrow i$	modulus in division
`& | ^ « »`	$i \times i \longrightarrow i$	bitwise and, or, xor, shifts
`== != <= < >= >`	$f \times f \longrightarrow$ `int`	comparisons

4.8.1 Implicit Type Conversions

If the types of an operators operands do not match, C performs a number of implicit conversions on the operands before applying the operator. Consider the following example:

```
int x;
double d;
...
d = d + x;
```

Open in Browser

The addition operator is not defined for a `double` and an `int` (see Table 4.8.1). Following the hierarchy shown below, C converts the `int` operand to a `double` value.

$$\begin{tikzcd}
\texttt{short} \arrow[dr] & & & & \\
& \texttt{int} \arrow[r] & \texttt{long} \arrow[r] & \texttt{float} \arrow[r] & \texttt{double} \\
\texttt{char} \arrow[ur] & & & &
\end{tikzcd}$$

Figure 4.8.2 Implicit type conversion for operands of arithmetic operators. All operands are converted to the least common type greater than or equal to `int`.

The implicit conversions adjust the operands to the least common operand type, where a "less than" $b \iff a \longrightarrow^* b$. Additionally, no operation is performed with operand types "less than" `int`. The motivation for this choice is that an `int` has usually the width of a machine word, for which the fastest operations of the processor are designed.

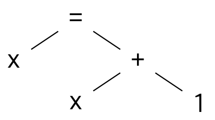
Remark 4.8.3 When such conversions are applied, the compiler needs to emit machine code that transforms a bit sequence representing the `int` into a bit sequence representing a `double` of the same numerical value. For some types (like 64-bit `long` to 64-bit `double`), this conversion is not exact due to the `double`'s limited precision.

4.8.2 L- and R-Evaluation of Expressions

In C (and derived imperative languages) there are two different ways used to evaluate expressions, depending on the position of the expression. Consider the assignment

```
x = x + 1;
```

Open in Browser



In C, assignments are expressions, meaning that = is just a binary operator. As a side effect (see Table 4.8.4), the operator stores the value resulting from the right-hand-side expression to the container whose address results from evaluating the expression of the left-hand side.

Consequently, the left operand of = needs to be evaluated as an address, whereas the right one should yield a value. In our example, a subexpression with the variable name `x` occurs on both sides of the operator. On the right-hand side, the subexpression `x` is evaluated by reading the content of the container bound to `x`. This evaluation to the container content is called **R-evaluation**. The resulting value is than incremented by `1` to compute the value of the right operand of =. On the left-hand side, `x` is evaluated as the *address* of the container bound to the variable `x`. We call this **L-evaluation**, as it occurs on the left side of the assignment.

Every expression can be R-evaluated, but some cannot be L-evaluated. To be L-evaluable, an expression needs to refer to the address of a container. For example, `1 + 2` cannot be L-evaluated, whereas `x` (where `x` is a declared variable) can. We can statically decide whether an expression is L-evaluable. The compiler aborts the translation with an error message if it finds an expression that is not L-evaluable at a position where L-evaluation is required.

If an expression e is L-evaluable, R-evaluating e results in the content of the container referred to by the L-evaluation of e.

4.8.3 Side Effects

A particular quirk of C is that evaluating expressions can have side effects. During expression evaluation, the contents of containers may be changed, the program can interact with the operating system (for example to print text on the screen, to send network packets, to read or write files, etc.), or the evaluation might even diverge (i.e. never terminate). Table 4.8.4 shows all C operators with side effects.

Table 4.8.4 Operators with side effects. ⊕ represents an arbitrary binary operator (see Table 4.8.1). The expression e needs to be L-evaluable for these operators.

Expression	Name	Value of the Expression	Side Effects
`++e`	pre-increment	new value of `e`	write `e + 1` to `e`
`--e`	pre-decrement	new value of `e`	write `e - 1` to `e`
`e++`	post-increment	old value of `e`	write `e + 1` to `e`
`e--`	post-decrement	old value of `e`	write `e - 1` to `e`
`e1 = e2`	assignment	new value of `e1`	write `e2` to `e1`
`e1 ⊕= e2`	assignment	new value of `e1`	write `e1 ⊕ e2` to `e1`
`f(...)`	function call	`f`’s return value	`f`’s side effects

The assignment is an expression with side effects:

```
x = x + 1
```

The side effect is that the value evaluated from the right-hand side is written to the container referred to by the left-hand operand. The written value is also the value of the expression.

If an expression has more than one side effect, different evaluation orders might lead to different results. Consider the following example:

```
x = 5;
y = (x = 2) * (++x);
```

The meaning of this program could be defined in several ways:

1. A non-deterministic semantics, meaning that both, `2` and `3`, would be correct results for the contents of `x` after the statements.

2. With a defined order in which the subexpressions are to be evaluated (as it is done in the programming language Java).

3. With no semantics at all (as it is done in C). The exact rules on when an expression with side effects has a semantics in C is complicated. As a rule of thumb, we can assume the evaluation of an expression to be undefined if it writes more than once into the same container. This does not extend to side effects in called functions:

   ```
   int a(void) { printf("A"); return 1; }
   int b(void) { printf("B"); return 2; }
   int c(void) { return a() + b(); }
   ```

   Every call to `c` returns the value `3`. However, either of `AB` and `BA` are correct outputs of the program. Since the rules for side effects in C are complicated and easy to misunderstand, it is best to reduce side effects to a minimum: At most one operator with side effects per expression.

   Aside: see Section J.2 and Section 6.5 in the C Standard.

4.8.4 Lazy Evaluation

C has three operators whose operands are evaluated lazily rather than strictly. An operator is subject to **strict** evaluation, if *all* its operands are evaluated before its value is computed. If it is evaluated lazily, operands are only evaluated if they contribute to the result.

Aside: In the presence of side effects, lazy or strict evaluation makes a difference: If an operand is not evaluated due to lazy evaluation, its side effects do not take effect.

Table 4.8.5 Operators that are evaluated lazily

Operator	Signature	Behavior
`e ? t : f`	int $\times T \times T \to T$	if `e` evaluates to `0`, evaluate `f`, and `t` otherwise
`e1 && e2`	int $\times$ int $\to$ int	`e1 == 0 ? 0 : e2`
`e1 || e2`	int $\times$ int $\to$ int	`e1 != 0 ? 1 : e2`

4.8.5 Exercises

Checkpoint 4.8.6 Bit Operators. The goal of this exercise is to learn how bit operations that you have encountered in the previous sections are realized in C.

C	Arithmetic
`a & b`	$a \mathbin{\&} b$
`a ^ b`	$a \mathbin{\hat{}} b$
`a | b`	$a \mid b$
`a « b`	$a \ll b$
`a » b`	$a \gg b$
`~a`	$\overline{a}$

Write a functions with the prototype

```
unsigned int f(unsigned int k, unsigned int n);
```

1. that flips the k-th bit in n if k is between 0 and 31.
2. that sets the k-th bit in n if k is between 0 and 31.
3. that clears the k-th bit in n if k is between 0 and 31.
4. that rotates n by k to the left if k is between 1 and 31.
5. that rotates n by k to the right if k is between 1 and 31.

You may assume here that `unsigned int` represents 32-bit unsigned integers.
Solution.

```c
unsigned int f1(unsigned int n, unsigned int k) {
  assert(k < 32);
  unsigned int a = 1 << k;
  return n ^ a;
}

unsigned int f2(unsigned int n, unsigned int k) {
  assert(k < 32);
  unsigned int a = 1 << k;
  return n | a;
}

unsigned int f3(unsigned int n, unsigned int k) {
  assert(k < 32);
  unsigned int a = (unsigned int)1 << k;
  return n & ~a;
}

unsigned int f4(unsigned int n, unsigned int k) {
  assert(0 < k && k < 32);
  unsigned int s = 32;
  return (n << k) | (n >> (s - k));
}

unsigned int f5(unsigned int n, unsigned int k)
{
  assert(0 < k && k < 32);
  unsigned int s = 32;
  return (n << (s - k)) | (n >> k);
}
```

Open in Browser

4.9 Calling Functions

Calling a function is also an expression in C. Review our introducing example, especially the call of `fac` in `main`:

```c
int fac(int n) {
    int res = 1;
    /* ... */
    return res;
}

int main(int argc, char** argv) {
    /* ... */
    int n = /* ... */;
    int r = factorial(n);
    /* ... */
}
```

Open in Browser

How exactly are parameters passed to `factorial`? When a function is called, a container for each parameter with the appropriate size is allocated. Practically, the parameters are additional local variables of the called function. Their scope extends over the entire function and their life time ends when the function is left. Before execution continuous with the first statement of the function, all (comma-separated) **arguments** are R-evaluated. The resulting values are written to the corresponding containers. The parameters' containers are deallocated when the function returns. This approach is called **call by value**, since the values of the arguments are passed to the called function. Other programming languages (e.g. Pascal or C#) additionally allow to **call by reference**, that is to pass the containers' addresses to the function. Arguments that are passed by reference need to be L-evaluable. In C, we can simulate passing arguments by reference if we declare the function to take a parameter of pointer type (see Section 4.10).

Aside: A function's prototype assigns a type to each parameter. As the prototype needs to be known to the compiler at the call site, it can determine the right container size.

A function terminates when a return-statement is executed. If the function has a `void` return type, we can omit the return-statment and the execution will return to the caller after the last statement of the called function's body. If the program has a different return type, it needs to be ended with `return E;`, where `E` is an expression with the appropriate type.

4.10 Pointers

A **pointer** is a variable with a type `T*` derived from a referenced type `T`. The value of a pointer is an address. Therefore, the content of a pointer's container is the address of another container. It is important to not confuse a pointer's value (which is an address) with the address of its container.

Table 4.10.1 Overview of C's pointer operators

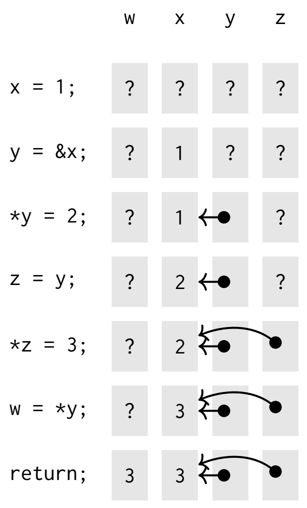
Operator	Name	Evaluation Rule	L-evaluable	Operand has Type	Expression has Type
`*`	indirection	L-eval($*e$) := R-eval(e)	yes	$T*$	T
`&`	address of	R-eval($\&e$) := L-eval(e)	no	T	$T*$

There are two operators that are particularly relevant for dealing with pointers: The address-of operator & and the indirection operator *. In a nutshell, R-evaluating &e gives the address of the L-evaluable expression e. *e on the other hand makes an L-evaluable expression out of e, if e R-evaluates to an address. Thus, *e denotes the content of the container referred to by the address e. Table 4.10.1 summarizes the properties of both operators.

```c
int w;
int x;
int *y;
int *z;
```

Open in Browser

(a) Variable Declarations



(b) Execution Trace

Figure 4.10.2 Execution trace of a C program with pointers. The declarations for the local variables are shown on the left. Every line in the trace corresponds to a statement, shown next to it. Each row displays the variable contents *before* executing the corresponding statement.

The program in Figure 4.10.2 shows how the operators * and & are used. Let us look at the effects of the individual instructions in detail:

1. The value 1 is written into the container of x.

2. The container of y can carry addresses of int containers. The expression x is L-evaluable: The result is the address of the container of x. R-evaluating the address-of operator therefore also results in this value. This address is then written to the container of y. Without the address-of operator, the compiler would find a type error since expressions on the right side of an assignment are R-evaluated, and the R-evaluation of x reads the value from the container of x and does *not* provide its address.

3. R-evaluating the expression 2 gives the value 2. The left side of the assignment is more exciting: The expression *y is, by definition, L-evaluable; otherwise it could not occur on the left side of an assignment. Its value is the result of the R-evaluation of its operand. Here, this is the content of the container of y: the address of x. Therefore, the value 2 is written to the container bound to x.

4. L-evaluating z results in the address of the container of z, whereas R-evaluating y gives the content of the container of y, which is the address of x. Consequently, the address of x is written to the container of z.

5. As z contains the address of x, the content of x's container is now changed to 3.

6. The interesting part is the occurrence of the dereference operator on the right side: R-evaluating `y` yields the content of `y`'s container, i.e. the address of `x`. We have seen that the expression `*y` is L-evaluable. The R-evaluation of an L-evaluable expression just reads the contents of the container whose address results from the L-evaluation. In this example, that is the content of the container of `x`, 3. In the end, `w` therefore carries the value 3.

4.10.1 Arrays

So far, our examples mostly contained only scalar types. A container for a variable with scalar type can carry exactly one value of the type at a time. If we want to process more data, for example a list of n numbers, we need aggregate data types. A value of an aggregated type consists of multiple (sub-)values of other (possibly also aggregated) types. We consider two kinds of aggregate types in C: **arrays** and **structs** (sometimes also referred to as **record**). We will discuss structs in Section 4.13.

Aside: There is another aggregated type in C: `union` types. We do not discuss this one here for brevity.

An array with size n and type T can carry n values of the type T. The following code declares an array of 100000 elements of type `int` and allocates *one* container that can carry 100000 `int` values:

```c
int a[100000];
/* ... */
int m = max(a);
```

Open in Browser

A peculiarity of C-arrays is that R-evaluating an array `a` of type `T[n]` does not provide a value of this type, but the array's base address. This is a pointer to the first element of `a`, which has the type `T*`.

Remark 4.10.3 A construct that seems like an exception to this rule is the occurrence of a variable with an array type in `sizeof`. Assuming that `sizeof(int)` = 4, `sizeof(a)` will be the value 400000 instead of `sizeof(int*)`. The `sizeof` operator does not (R-)evaluate its operand and is therefore not affected by this implicit conversion.

For this reason, in a function that expects an array of elements of type T, the corresponding parameter needs to have the type *pointer* to T. Since an array only contains its content and not its length, the length needs to be provided separately:

```c
int max(int* numbers, int len)
{
    int m = 0;
    for (int i = 0; i < len; i++)
        m = numbers[i] > m ? numbers[i] : m;
    return m;
}
```

Open in Browser

We can access the individual elements of an array via pointer arithmetic relative to the base address. As illustrated in the above example, we use the subscripting operator `e[i]` for this purpose. The expression `e[i]` is however only "syntactic sugar" for the expression `*(e + i)`. If `e` has type `T*`, we can add an expression `i` of type `int` to `e` and obtain another pointer value of type `T*`, offset from `e`. With this construct, we can create different pointers into the same container. Unfortunately, it is easy to construct invalid pointers with pointer arithmetic: For example, with the above declaration of the array `numbers`, the addresses `numbers + 100042` and `numbers - 1` are invalid. The corresponding offsets (100042 and −1) are outside the limits [0; 100000[implied by the size (100000) of the container. Constructing (not only accessing!) such "out-of-bounds" pointers causes undefined behavior (Section 4.15).

Remark 4.10.4 It is a common beginner's mistake to use `sizeof(numbers)` to obtain the array's length when `numbers` is only a *pointer* to the first element. As the value of `sizeof` is determined during the compilation based on the type of the given expression, it will provide the size of a pointer.

Remark 4.10.5 An exception is the pointer value `p+S` where `S` is the size of the container with base address `p`. This address can be computed without undefined behavior, but accessing a value at this address is undefined. The motivation for this exception is to allow iterating over arrays by incrementing a pointer into the array until this address is reached.

4.10.2 const

C provides means to decorate types with so-called **qualifiers**. A qualifier that is commonly used with pointers is `const`. The container of a variable with a `const`-qualified type cannot be overwritten (which is enforced by the compiler). For example:

```c
int const x = 2;
x = 4; /* Forbidden! */
```

Open in Browser

For pointers, there are several variants of using `const`, controlling what exactly cannot be overwritten:

```c
int w = 0;
int x = 1;
int const* p = &x;        /* pointer to a const int */
w  = *p;                  /* allowed */
*p = w;                   /* forbidden */
p  = &w;                  /* allowed, since the pointer is
    not const */
int * const q = &x;       /* const pointer to an int */
*q = 1;                   /* allowed, as the pointed-to
    value is not const */
q  = &w;                  /* forbidden, since the pointer is
    const */
int const *const r = &x; /* const pointer to const int */
*r = 1;                   /* forbidden */
r  = &w;                  /* forbidden */
```

Open in Browser

Listing 4.10.6 Different positions of `const` in pointer types

Big data structures are usually passed to functions as a pointer. It is common for functions to only read data from such a data structure. Such behavior can be documented by marking the corresponding parameter as a pointer to a `const` container in the function's prototype. The compiler enforces this behavior by not allowing writes to the corresponding container.

For example, we can now implement a `max` function that finds the maximal `int` in an array:

```c
int max(int const* table, unsigned len) {
  assert(len > 0 && "cannot␣take␣the␣maximum␣of␣an␣empty␣
        array");
    int max = table[0];
    for (unsigned i = 1; i < len; i++) {
        int v = table[i];
        max = v > max ? v : max;
    }
    return max;
}
```

Open in Browser

With the assertion in the first line, we validate that the function is not called with inputs that the remaining code does not expect. The string connected to the condition on `len` with the logical "and" operation is meaningless for the assertion, but it is displayed in the output of the program if it terminates because the assertion is violated in an execution.

4.10.3 Exercises

1. **C Types and Strings.**
You can solve this exception online at https://cdltools.cs.uni-saarland.de/prog2interpreter/chapter/advanced_c?num=2.

Write a C function that computes and prints the Caesar encryption of a string with an offset of *n*. The Caesar encryption replaces each letter of the message with the one *n* places later in the alphabet. For letters at the end of the alphabet, the offset wraps around to the beginning of the alphabet.

As arguments, the function obtains *n* and the character string to be encrypted. You may assume that the string only contains the ASCII symbols `a-z`.

What is different from and what is similar to an implementation in assembly?

Solution.

```
#include <stdio.h>

int caesar(char* text, int n) {
  if (n > 25 || n < 0) {
    printf("The value of n needs to be between 0 and 25
        (was %i)", n);
    return 1;
  }

  while (*text != '\0') {

    // shift
    *text += n;

    // fix shifting boundary violations
    if (*text > 'z') {
      *text = *text - 'z' + 'a' - 1;
    }

    text++;
  }

  return 0;
}

int main() {
  int shift = 5;
  char text[] = "abcdz";

  printf("Plain␣text:\n");
  printf("%s\n", text);
  printf("Shift␣by␣%i\n", shift);

  int ret = caesar(text, shift);

  if (ret == 0) {
    printf("Cipher:\n");
    printf("%s\n", text);
  }

  return ret;
}
```

Open in Browser

2. **Pointers.**
What is the output of the following program? Track the pointers and values manually before you run the program.

```
#include <stdio.h>

int main(int argc, char* argv[])
{
  int *dr, *ds;
  int **drr, **dss;
  int i;
  int aa[5];

  for (i = 0; i < 5; i++) *(aa + i) = -123;

  dr = &aa[3];
  ds = dr - 3;
  drr = &dr;
  dss = &ds;
  *dr = 23;
  *ds = 11;
  *(dr + 1) = 5;
  ds += 1;
  *ds = 66;
  aa[2] = 42;
  printf("%d␣%d␣%d␣%d␣%d\n", aa[0], aa[1], aa[2],
      aa[3], aa[4]);

  aa[0] = *(aa+2) - *(aa + 1);
  *(*drr - 1) = 7;
  *(*dss + 3) = 4;
  **dss = **drr + *(*drr - 2);
  printf("%d␣%d␣%d␣%d␣%d\n", aa[0], aa[1], aa[2],
      aa[3], aa[4]);

  return 0;
}
```

Open in Browser

Solution.

```c
#include <stdio.h>

int main(int argc, char* argv[])
{
  // declare pointers to int objects
  int *dr, *ds;
  // declare pointers to pointers to int objects
  int **drr, **dss;
  int i;
  // declare an array of 5 ints
  int aa[5];

  // set every array element to -123
  for (i = 0; i < 5; i++) *(aa + i) = -123;

  // dr is assigned the address of aa's fourth element:
      dr-->aa[3]
  dr = &aa[3];
  // ds is assigned the address of aa's first element:
      dr-->aa[0]
  ds = dr - 3;
  // drr now points to dr: drr-->dr-->aa[3]
  drr = &dr;
  // dss now points to ds: dss-->ds-->aa[0]
  dss = &ds;
  // the container pointed to by dr is set to 23: aa[3]
      = 23
  *dr = 23;
  // the container pointed to by ds is set to 11: aa[0]
      = 11
  *ds = 11;
  // the container behind the one pointed to by dr is
      set to 5: aa[3+1] = 5
  *(dr + 1) = 5;
  // ds now points to the next array element:
      dss-->ds-->aa[1]
  ds += 1;
  // the container pointed to by ds is set to 66: a[1]
      = 66
  *ds = 66;
  aa[2] = 42;
  // Output: 11 66 42 23 5
  printf("%d␣%d␣%d␣%d␣%d\n", aa[0], aa[1], aa[2],
      aa[3], aa[4]);

  // a[0] = aa[2] - aa[1] = 42-66 = -24
  aa[0] = *(aa+2) - *(aa + 1);
  // the container before the one pointed to by dr is
      set to 7: a[3-1] = 7
  *(*drr - 1) = 7;
  // The content of the third container behind ds is
      set to 4: a[1+3] = 4
  *(*dss + 3) = 4;
  // aa[1] = aa[3] + a[3-2] = 23+66 = 89
  **dss = **drr + *(*drr - 2);
  // Output: -24 89 7 23 4
  printf("%d␣%d␣%d␣%d␣%d\n", aa[0], aa[1], aa[2],
      aa[3], aa[4]);

  return 0;
}
```

Open in Browser

4.11 Examples

We now have the means to implement the examples from Chapter 3 in C.

Example 4.11.1 Number Conversion.

```c
#include <stdio.h>

void print_hex(unsigned word)
{
    unsigned shift = 8 * sizeof(word);
    do {
        shift -= 4;
        unsigned digit = (word >> shift) & 0x0f;
        unsigned ascii = digit + (digit < 10 ? '0' : 'a' -
            10);
        putchar(ascii);
    } while (shift != 0);
}
```

Open in Browser

Listing 4.11.2 Number conversion implementation

□

Example 4.11.3 The Sieve of Eratosthenes.

```c
#include <string.h>

char* eratosthenes(char* table, unsigned n) {
    memset(table, 0, n);
    for (unsigned q = 2; q * q < n; q++) {
        if (table[q] == 0) {
            for (unsigned j = q * q; j < n; j += q)
                table[j] = 1;
        }
    }
    return table;
}
```

Open in Browser

Listing 4.11.4 Sieve of Eratosthenes implementation

□

Example 4.11.5 Insertion Sort.

```c
#include <assert.h>

static unsigned find(int* arr, unsigned n, int value)
{
    unsigned i = 0;
    while (i < n && arr[i] < value)
        i++;
    return i;
}

static void insert(int* arr, unsigned n, unsigned from,
    unsigned to)
{
    assert(to <= from);
    int val = arr[from];
    for (unsigned i = to; i <= from; i++) {
        int tmp = arr[i];
        arr[i]  = val;
        val     = tmp;
    }
}

void insertsort(int* arr, unsigned n)
{
    for (unsigned i = 1; i < n; i++) {
        unsigned pos = find(arr, n, arr[i]);
        insert(arr, n, i, pos);
    }
}
```

Open in Browser

Listing 4.11.6 Insertion sort implementation

□

4.12 Dynamic Memory Management

The life time of a local variable is bound to the execution of its surrounding block. Consider the following example:

```c
int *numbers(int n) {
    int a[n];
    for (int i = 0; i < n; i++)
        a[i] = i;
    return a;
}
```

Open in Browser

Apparently, it attempts to return an array with the first n non-negative integers. However, the life time of `a` is restricted to the execution of the body of the `numbers` function, since it is the innermost block that surrounds the declaration. When the execution returns from `numbers` to the caller, the container of `a` is thus deallocated. The corresponding pointer to the first element of the array *dangles*: It points to no existing container. Therefore, the program is incorrect. There are two ways to solve this problem:

1. We pass the address of a previously allocated result array to `numbers`. The

function is then no longer concerned with allocating a container for the resulting array. Most of the time, this solution is to be preferred since it is more flexible. The array could (but does not need to) be declared as a local variable in the calling function.

2. The `numbers` function can itself allocate a container for the array. Since a local variable is not fit for this task, a container on the **heap** is the only option. These are allocate with the `malloc` function.

If we want containers that live longer than the execution of their surrounding block, we need to allocate them with the `malloc` function:

```
void* malloc(size_t  n)
```

Open in Browser

The parameter `n` determines the size of the allocated container in bytes (its type `size_t` is provided by the compiler and defined to be a sufficiently large unsigned integer type). It is very common to use the `sizeof` operator to define the container's size in terms of the type of its intended content. For example, the following statement allocates an array of `n` elements of the type `T`:

```
T *a = malloc(n * sizeof(a[0])); // sizeof(a[0]) ==
    sizeof(T)
```

Open in Browser

Aside: If we want to use `malloc` to allocate an array to be referred to by a pointer `a` of type `T`, it is better to use `sizeof(a[0])` rather than `sizeof(T)`. If we later on change the type of `a`, we do not need to change it in the `sizeof` expression as well.

Remark 4.12.1 `sizeof(x)` is an operator, not a function. Its operand `x` can either be a type or an expression. If it is an expression, the *compiler* computes the expression's type and determines its size. `sizeof` is not evaluated during the program's execution and does therefore not provide the size of a container, but of a type.

Such a container lives until the pointer returned by `malloc` is given to the `free` function for deallocation:

Aside: "double frees", deallocations of already deallocated containers, are common programming errors that are not always easy to discover.

```
void free(void* b)
```

Open in Browser

In contrast to the containers of local variables, `malloc`ed containers are not deallocated when execution leaves the block that surrounds their allocation.

Example 4.12.2 Eratosthenes. Consider a `main` function that obtains the size of the table for the Sieve of Eratosthenes from a command line parameter:

```
int main(int argc, char** argv) {
    if (argc < 2) {
        printf("syntax:␣%s␣<n>\n", argv[0]);
        return 1;
    }
    unsigned n = atoi(argv[1]);
    char *table = malloc(n * sizeof(table[0]));
    eratosthenes(table, n);
    print_table(table, n);
    free(table)
    return 0;
}
```

Open in Browser

□

4.12.1 realloc

A common use of arrays is to implement lists. This is particularly sensible if we need fast access to arbitrary elements: The address of the `i`-th element of an array can be computed using pointer arithmetic.

The size of an array is determined at its creation (at the declaration for a local variable, or as an argument to the `malloc` call) and cannot be changed retroactively. If the number of elements that are added to an array-based list exceeds the array's size, a new, larger array needs to be allocated, the previous contents copied, and the old array needs to be deallocated. The `realloc` function performs all these steps:

```c
int* read_numbers_from_file(FILE *f) {
    int *numbers = NULL;
    unsigned capacity = 0;
    unsigned n = 0;
    int x;
    while (fscanf(f, "%d", &x) == 1) {
        if (n == capacity) {
            capacity = capacity == 0 ? 1 : 2 * capacity;
            numbers = realloc(numbers,
                sizeof(numbers[0])*capacity);
        }
        numbers[n++] = x;
    }
}
```

Open in Browser

When implementing such an array list, it is sensible to double the array's size with every `realloc` call: When n elements are inserted, the space for at most $n-1$ elements is wasted while at most `2n` copies need to be performed.

4.13 Structs

The second aggregate data type in C that we consider is the **struct**. Structs are also commonly referred to as **record**. A struct has n **fields** of possibly distinct types $T_1, \ldots, T_n$. A container for a struct is the combination of containers for the n fields. For an example, consider a struct that represents aspects of a person. We want to consider for each person a date of birth, a first name, and a last name.

A date consists of a day, a month, and a year:

```c
struct date {
    unsigned day, month, year;
};
```

Open in Browser

We can use **composition** ("is part of") or **aggregation** ("refers to") to define our struct:

```c
struct person {
    struct date
        date_of_birth;
    char name[16];
    char surname[16];
};
```

Open in Browser

Listing 4.13.1 Composition

```c
struct person {
    struct date
        date_of_birth;
    char const *name;
    char const *surname;
};
```

Open in Browser

Listing 4.13.2 Aggregation

In both declarations, the `date_of_birth` field is added through composition. We could also implement the field via aggregation with a pointer to a `date` struct. However, the `struct date` is a small struct, therefore referencing it with a pointer saves little space in the `person` struct. Reusing `struct date` objects provides therefore little benefit for this case.

We can access the individual fields of a struct container with the `.` operator:

```c
struct person p;
/* ... */
printf("%s␣%s\n", p.name, p.surname);
```

Open in Browser

In contrast to arrays, structs can be passed to functions by value in C. With small structs, this can be sensible. In practice however, we usually pass structs via pointer as in the following example:

```c
void init_date(struct date *d, unsigned day,
                unsigned month, unsigned year)
{
    d->day = day;
    d->month = month;
    d->year = year;
}
```

Open in Browser

The expression `A->B` is an abbreviation for `(*A).B`.

Remark 4.13.3 In C, suffix operators have higher *precedence* than prefix operators. Therefore, `*A.B` is not the same as `(*A).B`: The former accesses a struct field and loads from the address contained there, the latter loads the field of a struct whose address is given. In practice, it is best to use parentheses to make non-obvious precedences explicit.

4.13.1 typedef

Types in C can easily become long, like `struct person`, or complex, like `int *(*)[10]`. C provides the means to define shorter, more meaningful names for such types:

```c
typedef struct {
    /* see above */
} person_t;

/* ptr_table_t ist a pointer to an array
    of 10 pointers to ints */
typedef int *(*ptr_table_t)[10];
```

Open in Browser

Example 4.13.4 Initialization. It is a common pattern to define for each struct a function that properly initializes it. The following function initializes the version of `struct person` where `name` and `surname` are aggregated:

```
person_t *person_init(person_t *p, date_t const *d,
                      char const *name, char const *surname)
{
    p->date = *d;
    p->name = name;
    p->surname = surname;
    return p;
}
```

Open in Browser

Such initialization functions, which are also called **constructors**, usually expect a pointer to an uninitialized struct container. This makes the constructor independent of how the struct is allocated: as a local variable, a global one, or dynamically:

```
person_t global_p;
void person_test() {
    person_t local_p;
    person_t *heap_p;

    heap_p = malloc(sizeof(*heap_p));

    person_init(&global_p, /* ... */);
    person_init(&local_p, /* ... */);
    person_init(heap_p, /* ... */);

    free(heap_p);
}
```

Open in Browser

If we incorporate the `name` and `surname` fields by composition instead of aggregation, we need to copy the character sequences into the struct:

```
person_t *person_init(person_t *p, date_t const *d,
                      char const *name, char const *surname)
{
    p->date = *d;
    snprintf(p->name,    sizeof(p->name),    "%s", name);
    snprintf(p->surname, sizeof(p->surname), "%s", surname);
    return p;
}
```

Open in Browser

It is important to not copy more characters than fit into the field. We realize this here with the `snprintf` function with the size `sizeof(p->name)`. If we copied the string without ensuring that the field size is respected, we would encounter undefined behavior (see Section 4.15) when the function is called with too large strings. We can use `sizeof` here because the type of `p->name` here is `char[16]` and not `char *`. The value of `sizeof(p->name)` is therefore 16, rather than the size of a pointer. □

Example 4.13.5 Polynomials. The following struct represents a polynomial of degree `n`:

```c
typedef struct {
  unsigned degree;
  int *coeffs;
} poly_t;
```

Open in Browser

The coefficients are implemented through aggregation. This is necessary, since we do not want to assume a statically known fixed length for the polynomials. Therefore, an array for the coefficients need to be allocated separately (see Exercise 4.13.3.1). □

4.13.2 Incomplete Structs

We use a technique called **encapsulation** to separate large software systems into modules. By encapsulation, we hide the implementation details of a module from code that uses it. The benefit of this technique is that we can then change the module's implementation without affecting the code parts that use the module.

C supports encapsulation through incomplete structs. They allow to hide the specific structure of a struct from other code. We will discuss encapsulation in more detail in the Java part of this book.

Consider Example 4.13.5. Let us assume that we want to hide how polynomials are represented from using code parts. We achieve this with an incomplete struct that we declare in a header file (see `poly.h` in Figure 4.13.6). The values of the data type are then only accessible through a set of functions, a so-called *Application Programming Interface* (API). The API of such a data type only uses pointers to the incomplete type as the encapsulation ensures that its contents, and therefore the necessary container size, are not known to the outside.

```c
#ifndef POLY_H
#define POLY_H

/* incomplete struct */
typedef struct poly_t poly_t;

poly_t *poly_alloc(unsigned degree);
void poly_free(poly_t *p);
void poly_set_coeff(poly_t *p, unsigned deg,
                    int coeff);
int poly_eval(poly_t const *p, int x);
unsigned poly_degree(poly_t const *p);

#endif /* POLY_H */
```

Open in Browser

(a) poly.h

```c
#include "poly.h"

struct poly_t {
    unsigned degree;
    int *coeffs;
};
```

Open in Browser

(b) poly.c. Continuation: Exercise 4.13.3.1

```c
#include <stdlib.h>
#include <stdio.h>
#include "poly.h"

int main(int argc, char *argv[])
{
    if (argc < 3) {
        fprintf(stderr, "syntax:␣%s␣x␣coeffs...",
                argv[0]);
        return 1;
    }

    poly_t *p = poly_alloc(argc - 3);
    for (int i = 2; i < argc; i++) {
        int coeff = atoi(argv[i]);
        poly_set_coeff(p, i - 2, coeff);
    }
    int x = atoi(argv[1]);
    int y = poly_eval(p, x);
    poly_free(p);
    printf("%d\n", y);
    return 0;
}
```

Open in Browser

(c) main.c

Figure 4.13.6 Implementation of polynomials with encapsulation. The internal structure of the struct poly_t is only visible in the translation unit poly.c.

4.13.3 Exercises

1. **Polynomial.**
   We consider the API for polynomials defined in Figure 4.13.6. Complete the file `poly.c`: Implement the functions declared in the header file `poly.h` in.

   (a) Implement a function `poly_alloc` allocating the space for a polynomial on the heap.

   (b) Implement the function `poly_free`, that frees a polynomial that was constructed on the heap.

   (c) Implement the function `poly_set_coeff` that sets the coefficients of an polynomial.

   (d) Implement the evaluation of a polynomial using the Horner schema.

   (e) Implement the function `poly_degree` that returns the degree of a polynomial.

   Solution.

```c
#include <stdlib.h>
#include <assert.h>

#include "poly.h"

struct poly_t {
    unsigned degree;
    int *coeffs;
};

poly_t *poly_alloc(unsigned degree) {
    poly_t *p = malloc(sizeof(*p));
    p->degree = degree;
    p->coeffs = malloc((degree + 1)
                  * sizeof(p->coeffs[0]));
    return p;
}

void poly_free(poly_t *p) {
    free(p->coeffs);
    free(p);
}

void poly_set_coeff(poly_t *p, unsigned i,
                    int val) {
    assert(i <= p->degree);
    p->coeffs[i] = val;
}

int poly_eval(poly_t const *p, int x) {
    int res = p->coeffs[0];
    for (unsigned i = 1; i <= p->degree; i++)
        res = res * x + p->coeffs[i];
    return res;
}
```

Open in Browser

4.14 Input and Output

The header file `stdio.h` of the *C Standard Library* provides elementary functions for input and output. A file is here represented by a pointer to a `FILE`. The struct `FILE` is opaque (i.e. encapsulated away from us), we therefore do not know its internal structure and only use pointers to it.

`stdio.h` defines three global variables:

```c
FILE *stdin;
FILE *stdout;
FILE *stderr;
```

Open in Browser

They are called **standard input**, **standard output**, and **error output**.

Files can be opened with

```c
FILE* fopen(char const *filename, char const *mode);
```

Open in Browser

and closed again with

```c
void fclose(FILE*);
```

Open in Browser

`fopen` requires a file name and a mode, which specifies how the file is opened (`"r"` for reading, `"w"` for writing). It produces a pointer to a `FILE` if the file was successfully opened or `NULL` otherwise. For reading and writing, we often use the functions

```c
int fscanf(FILE* f, char const* format, ...);
int fprintf(FILE* f, char const* format, ...);
```

Open in Browser

The commonly used functions `printf` and `scanf` are abbreviations for these where `f` is set to `stdin` or `stdout`, respectively.

4.14.1 Format Strings

Both, `fprintf` and `fscanf` use so-called **format strings** to determine how input and output is formatted. The chosen format element also describes how the corresponding argument is to be interpreted. In the following, we give a few examples of format strings and refer to the corresponding Unix manpages for a complete documentation.

Aside: Type `man fprintf` in a Unix shell.

```c
char const *weekday = "Wednesday";
char const *month = "March";
int day = 11;
int hour = 10;
int min = 30;
printf("%s,␣%s␣%d,␣%.2d:%.2d\n",
    weekday, month, day, hour, min);
```

Open in Browser

produces this output:

```
Wednesday, March 11, 10:30
```

Floating point numbers can be printed as follows:

```c
printf("pi␣=␣%.5f\n", 4 * atan(1.0));
```

Open in Browser

resulting in:

```
pi = 3.14159
```

Example 4.14.1 Running Average. We want to write a program that computes the average of all numbers from a list of files. A technique to compute the average of a list of numbers is the *running average*. If we know the average s_n of the first n numbers $a_1, \ldots, a_n$, the average of the numbers $a_1, \ldots, a_{n+1}$ can be computed as:

$$s_{n+1} = \frac{n}{n+1} s_n + \frac{1}{n+1} a_{n+1} \tag{4.1}$$

We start with a function that computes the average of all numbers given in *one* file. We assume that the input files contain textual representations of floating-point numbers that are separated by whitespace characters as in this example:

```
1 2.2 3
4.2
1.4
```

Open in Browser

We convert the textual representation to a `double` floating-point value using `fscanf` and store it into a variable `val`. The average and the number of the read numbers are stored in a struct:

```
typedef struct {
    double avg;
    unsigned n;
} avg_t;
```

Open in Browser

The function `running_average` expects a file and a pointer to such a struct as arguments. It then reads as long as it can find numbers in the file and continues the average computation according to (4.1):

Aside: `fscanf` returns the number of correctly decoded elements of the format string, providing us a termination criterion.

```
avg_t *running_average(FILE* f, avg_t* avg)
{
    double val;
    while (fscanf(f, "%lg", &val) == 1) {
        double   old_n = avg->n;
        unsigned new_n = ++avg->n;
        avg->avg = (old_n / new_n) * avg->avg + val / new_n;
    }
    return avg;
}
```

Open in Browser

The following main function accepts a list of file names as command line parameters. It opens these files one after the other and calls the `running_average` function with it. Lastly, it prints the computed average. If no command line parameter is given, the standard input `stdin` is used as an input file.

```
int main(int argc, char *argv[])
{
    avg_t avg;
    avg.avg = 0.0;
    avg.n   = 0;

    if (argc == 1) {
        running_average(stdin, &avg);
    }
    else {
        for (int i = 1; i < argc; i++) {
            FILE* input;
            if ((input = fopen(argv[i], "r")) != NULL) {
                running_average(input, &avg);
                fclose(input);
            }
            else {
                fprintf(stderr, "no␣such␣file:␣%s\n",
                    argv[i]);
                return 1;
            }
        }
    }
    printf("%g\n", avg.avg);
    return 0;
}
```

Open in Browser

□

4.15 Undefined Behavior

Consider the following statement:

```
z = x / y;
```

Open in Browser

Let us assume that the variable x contains a pointer to a character and y an integer. This program is meaningless in this case, since C does not define the division of a pointer by a number. We call this kind of programming error a *type error*, as an operator is applied to values for whose type it is not defined. In typed languages like C, type errors are recognized statically (i.e. at compile time) during *type checking*. The type system allows the compiler to reject meaningless programs.

There are however meaningless programs that the compiler cannot reject. Assume for the same program that x and y are both integers, but y contains the value 0. The program is **well-typed** as integer division is defined in C. However, it is unclear what the execution should do in case of a division by 0.

There are essentially two solutions to this problem: The language could define a behavior for such exceptional situations, for example executing error-handling code. This way is pursued among others in Java, C#, ML in the form of throwing *exceptions*. The program then has a defined behavior for exceptional situations. However, this strategy requires that every operation that could potentially cause an exceptional situation is checked *at run time* for such a situation. This can slow down program execution *significantly*. We call a language **type safe** if every well-typed program cannot get "stuck", i.e. every input leads to an execution that is defined by the language. Robin Milner coined the following definition of type safety:

“Well-typed programs cannot go wrong.”

The other approach is to not define a behavior for exceptional situations. This is the way of C. If no behavior is defined for exceptional situations, the implementation does not need to check at run time whether one occurred. Therefore, C makes it potentially easier, in contrast to type-safe languages, to make programs run faster. However, the programmer has no guarantee what the execution does when an exceptional situation occurs. This can lead to severe errors that are very hard to identify. Annex J.2 of the C Standard [1] documents all situations where the behavior is undefined. Among many others, these include: Overflows of signed integer arithmetic, division by 0, forming out-of-bounds pointers, and accessing addresses that do not point to a container.

Aside: C therefore cannot count as a type-safe language. However, its type system still gives guarantees that help to rule out some kinds of run-time errors.

Let us deepen our understanding further by discussing an example of undefined behavior that violates **memory safety**, which means that it tries to access an address that does not point to any container. To this end, consider the following program:

```
void broken(int n) {
    int x[20];
    *(x + n) = 1;
}
```

Open in Browser

The program has defined behavior for $0 \leq n < 20$. For all $n \geq 20$, the behavior of the program is undefined, since the address resulting from `*(x + n)` does not point to the container (C99 Standard, Page 492):

“The behavior is undefined in the following circumstances: [...] Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that points just beyond the array object and is used as the operand of a unary * operator that is evaluated (6.5.6).”

That does not necessarily mean that the execution crashes or terminates with an error. It is just not defined what happens. Every assumption on an execution with undefined behavior is *false*. It is very well possible that the execution proceeds and produces the expected result. This makes it hard to find programming errors like the one in the `broken` function.

So why do we even bother with undefined behavior? Let us think about what the C compiler needs to do to translate `broken` into a machine language. If the behavior of an access with $n \geq 20$ should be defined, this condition needs to be checked. Therefore, the compiler needs to either prove (through static analysis of the code) that $n < 20$ holds for all possible inputs of the program. This is a difficult problem that, compilers generally cannot solve completely in practice. Otherwise, the compiler needs to emit code that checks the condition at the program’s run time:

Aside: This code assumes that `sizeof(int) == 4`.

```
    sltiu  $t1  $a0  20
    beqz   $t1  fail
    sll    $t1  $a0  2
    addu   $t0  $t0  $t1
    li     $t1  1
    sw     $t1  ($t0)
fail:
    ...
```

Open in Browser

The first two instructions only serve to recognize and handle the exceptional situation that the offset is outside of `x`. If such tests are, e.g., in the body of deeply nested loops, they can have a severe impact on the program's performance.

C's philosophy is to not enforce these checks, but rather give the responsibility to avoid undefined behavior to the programmer. In case the program that calls `broken` is correct, the test is unnecessary and wasting execution time can be avoided. Since the behavior in case $n \geq 20$ holds is undefined, the compiler can simply ignore this case: The emitted code only needs to be correct if $0 \leq n < 20$. The compiler can therefore emit code without the check:

```
sll   $t1 $a0 2
addu  $t0 $t0 $t1
li    $t1 1
sw    $t1 ($t0)
```

Open in Browser

If the programmer is not certain that the code calling `broken` guarantees that $0 \leq n < 20$ holds, they need to implement this check themselves:

```
void broken(int n) {
    int x[20];
    if (0 <= n && n < 20) {
        *(x + n) = 1;
    } else {
        // ...
    }
}
```

Open in Browser

4.15.1 Unspecified and Implementation-Defined Behavior

The C Standard clearly distinguishes between undefined, unspecified, and implementation-defined behavior.

Implementation-defined behavior represents non-determinism: There are multiple, clearly defined possibilities how the program may behave. The programmer can rely on the fact that it will follow one of these possibilities, but cannot assume which one. We have seen an example of unspecified behavior in Subsection 4.8.3: C does not (fully) specify the order in which the sub-expressions of an expression are evaluated. If the sub-expressions have side effects that do not intermingle, the program may have different behaviors. Annex J.1 of the C Standard documents the unspecified behavior.

Implementation-defined behavior refers to unspecified behavior where the compiler always selects a specific possibility. This choice is consistent for all programs that are translated with this compiler. Examples for implementation-defined properties are the maximal length of an identifier, the number-representation for signed integers (two's complement, one's complement, etc.), the size of the integer types, etc.. Annex J.3 of the C Standard documents the cases of implementation-defined behavior.

4.15.2 Security Problems

Negligence with undefined behavior can lead to errors that not only crash programs, but present severe security vulnerabilities. The classic example for this is the **buffer**

overflow. Assume an attacker has the binary code produced by a compiler for a C program.

They can now exploit unchecked stores into arrays (usually character strings) to manipulate the contents of the run-time stack. This way, they can change the program's control flow and ultimately inject malicious code into a program.

Consider the following example program:

```
void foo(char const *msg) {
    char nice_msg[32];
    sprintf(nice_msg, "The␣message␣is:␣\"%s\"", msg);
    ...
}
```

Open in Browser

The `sprintf` function is similar to `fprintf`, but it does not write the formatted string to a file or output, but rather into an array that is provided as the first parameter. The behavior is undefined if the array is not large enough to hold the entire resulting string.

Let us assume that the compiler emits the following MIPS code for `foo`:

```
addiu  $sp $sp -36
sw     $ra 32($sp)

# Code loading the address of
# "The messa ..." to $a1
move   $a2 $a0
move   $a0 $sp
jal    sprintf
...

lw     $ra 32($sp)
addiu  $sp $sp 36
jr     $ra
```

Open in Browser

The local variables of the C program are stored on the stack of the MIPS machine. For simplicity, we further assume that `foo` is called with an argument that is received via the network or read from mass storage, as it is common in practice.

The problematic part of `foo` is that it writes the provided string to a buffer with a fixed size (here 32) without checking whether the buffer overflows. What if `"The message is: "` concatenated with `msg` has more than 31 characters? Since `sprintf` only has the address of the destination array, the function does not know the array's size and can therefore not check against it. Therefore, it will potentially write to addresses behind the destination array in the machine's sequential memory. This destroys the consistency of the run-time stack and allows the attacker to overwrite the return address that has been saved on the stack by `foo`.

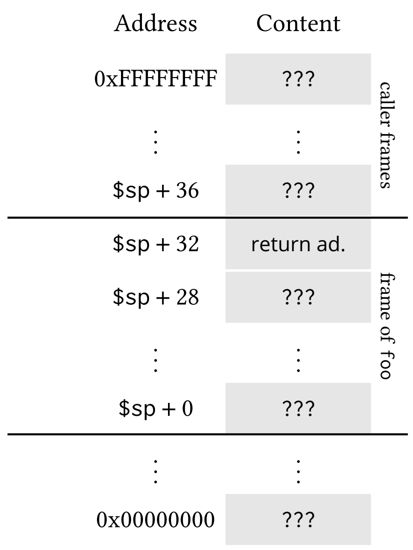


Figure 4.15.1 Memory excerpt of the run-time stack when calling `foo`

Assume the attacker knows the value of the stack pointer at the call. Then, they can fabricate a character string that inserts (almost) arbitrary code into the program. This code is part of the character string (as bytes of the binary encoded machine instructions). The attacker just needs to overwrite the return address with an address further up on the stack, where the first instruction to be executed is stored when the string is written to the stack.

When calling `foo` with a string of more than 15 characters, the behavior is undefined. Therefore, the compiler can ignore this exceptional situation and just emits machine code that implements the defined behavior. The machine code emitted by the compiler shown here has a behavior, but not the C program. The attacker in this example exploits this fact.

Such attacks depend on the way the program is compiled. In our example, the attacker needs to know the value of the stack pointer when `foo` is entered exactly to forge a fitting return address. Studying the program's binary code in detail is essential for this kind of attack.

By now, simple buffer overflow attacks can be mitigated by prohibiting that the compiler executes code that resides on the stack. Modern compilers can also sacrifice some execution speed to recognize overwritten return addresses. Attackers however continue to find ways to circumvent these mitigations. For example, the former mitigation can be avoided with so-called return-oriented programming[8], which does not inject new code to the stack, but misuses existing code snippets from the program itself for its malicious code. To this day, there is little effective protection from such attacks that does not affect the program's execution speed severely.

4.15.3 Undefined Behavior and the Compiler

Modern compilers use undefined behavior to optimize programs. As C only gives meaning to executions without exceptional situations, the compiler only needs to generate code that is correct for such executions. Conversely, the compiler can gain information for optimization opportunities from operations that can have undefined behavior.

Example 4.15.2 Unreachable Code Elimination. Consider the following program:

[8] `wikipedia.org`

```c
int x = y / z;
if (z == 0)
    return 1;
```

Open in Browser

The program's behavior is only defined if $z \neq 0$. If $z = 0$, its behavior is undefined because of a division by zero. The compiler can therefore assume that $z \neq 0$ and use this information to optimize the program.

According to the C semantics, it is therefore allowed for the compiler to eliminate the if-statement and to transform the program as follows:

```c
int x = y / z;
```

Open in Browser

□

It is common that such an optimization is misunderstood by programmers as "compiler bug". This misunderstanding has already lead to many errors in widely-used software systems, as we will see in the following.

4.15.4 Practical Examples

The following examples are taken from an article by Wang et al. [2]. More examples can be found there.

Example 4.15.3 Shifts. Let us consider a code fragment from the implementation of the ext4 file system in the Linux kernel:

Aside: Bug 14287, Linux kernel, 2009. `https://bugzilla.kernel.org/show_bug.cgi?id=14287`.

```c
...
groups_per_flex = 1 << s;
flex_group_count = ... / groups_per_flex;
```

Open in Browser

According to page 493 of the C99 Standard, the behavior of shifts is undefined in certain situations:

> "The behavior is undefined in the following circumstances: [...] An expression is shifted by a negative number or by an amount greater than or equal to the width of the promoted expression (6.5.7)"

Assume that `sizeof(int) == 4`. If the value of `s` is greater than or equal to 32, the programs behavior is undefined.

The C Standard leaves the behavior of such shifts undefined because different processors implement them differently. In the 32-bit versions of MIPS and the widely used x86 architecture, only the lowest 5 bit of the shift amount are taken into account. A shift by 32 there is equivalent to a shift by 0. On a PowerPC processor, the lowest *6* bit of the shift amount are considered. There, a left shift by 32 results in the value 0. Both implementations are justified. None is "wrong".

If C defined one of these meanings, or even a third one, for shifts, the implementations on *every* processor would need to respect this. For processors that implement shifts differently, this requires additional code. This is to be avoided according to C's philosophy, therefore the behavior is undefined to enable more efficient implementations for the *defined* behavior.

The above-mentioned bug report also noted that the code crashed on a PowerPC since `groups_per_flex` had the value 0, leading to a division by zero that caused an exception in the PowerPC processor. A programmer attempted to fix this error by testing the variable for 0 after the shift:

```c
...
groups_per_flex = 1 << s;
/* There are some situations, after shift the
    value of 'groups_per_flex' can become zero
    and division with 0 will result in fixpoint
    divide exception */
if (groups_per_flex == 0)
    return 1;
flex_group_count = ... / groups_per_flex;
```

Open in Browser

This correction is wrong, since it does not eliminate the undefined behavior. The C compiler that is used to compile the Linux kernel therefore promptly “optimized the fix away”. That is correct, since the program’s behavior is undefined when `s` is greater than or equal to the number of bits in an `int`. In the defined case, the content of `groups_per_flex` is never 0, thus the if-statement’s condition is always false and it can be removed.

A correct fix would be:

```c
if (s < 0 || s >= CHAR_BIT * sizeof(int))
    return 1;
groups_per_flex = 1 << s;
flex_group_count = ... / groups_per_flex;
```

Open in Browser

□

Example 4.15.4 Integer Overflow. The following code also originates from the Linux kernel. The type `loff_t` is a signed integer type.

```c
int do_fallocate(..., loff_t offset, loff_t len)
{
    struct inode *inode = ...;
    if (offset < 0 || len <= 0)
        return -EINVAL;
    /* Check for wrap through zero too */
    if ((offset + len > inode->i_sb->s_maxbytes)
        || (offset + len < 0))
        return -EFBIG;
...
```

Open in Browser

The code first ensures that both variables, `offset` and `len` are not negative. The condition `offset + len < 0` attempts to check whether the addition causes an overflow. However, C does not define the behavior of overflowing signed integer arithmetic. From the first if-statement, the compiler concludes that `offset` and `len` are not negative at the second if. Adding to non-negative integers either causes undefined behavior (if an overflow occurs), or results in a positive number. The compiler therefore optimizes the second if-statment as follows:

```c
...
if (offset + len > inode->i_sb->s_maxbytes)
    return -EFBIG;
...
```

Open in Browser

A correct test for the overflow could look as follows:

```
int do_fallocate(..., loff_t offset, loff_t len)
{
    struct inode *inode = ...;
    if (offset < 0 || len <= 0)
        return -EINVAL;
    /* Check for wrap through zero too */
    if ((LOFF_T_MAX - offset < len)
        || (offset + len > inode->i_sb->s_maxbytes)
        return -EFBIG;
...
```

Open in Browser

Here, LOFF_T_MAX needs to be the largest value contained in loff_t. Ideally, one writes a function for this check to make the code more readable and to concentrate potential errors to one place:

```
int loff_overflows(loff_t offset, loff_t len)
{
    assert(offset >= 0);
    assert(len > 0);
    return LOFF_T_MAX - offset < len;
}

int do_fallocate(..., loff_t offset, loff_t len)
{
    struct inode *inode = ...;
    if (offset < 0 || len <= 0)
        return -EINVAL;
    /* Check for wrap through zero too */
    if (loff_overflows(offset, len)
        || (offset + len > inode->i_sb->s_maxbytes)
        return -EFBIG;
...
```

Open in Browser

□

Chapter 5

Algorithms and Data Structures

In this section, we discuss several simple data structures and their implementation in C. We start with various list implementations, via arrays and pointer structures. Then, we briefly discuss binary trees.

5.1 Lists

One of the simplest data structures are lists. Lists represent an ordered collection of several elements in which, unlike a set, an element can appear multiple times. Typical operations of lists are:

- Get length
- Append element
- Remove element
- Get/set element at position

Lists can have different implementations. These implementations differ in how the list is represented (the data structure itself) and partly also in the run time complexity of the list operations.

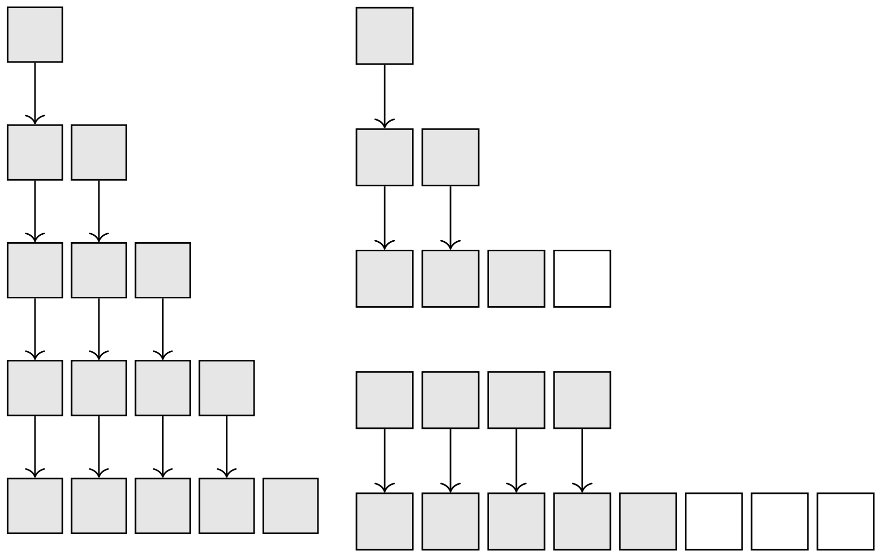
5.1.1 Array Lists

A very straightforward list implementation uses plain arrays. The idea is to put all elements in a list into an array. In addition to storing the elements in an array, we also need to memorize the length of the array because C arrays are just containers holding multiple elements and we cannot inquire a container for its size.

One decision that we have to make is how we handle resizing the array when elements are appended or removed. One way to handle this is to always match the length of the array with the length of the list. Then, for each append and each remove, the array has to be resized. As we discussed before, containers (remember an array is just a container) cannot change their size. To get a bigger container, we need to allocate a new one and copy the data there. So matching the array size to the length of the list would have the overhead of reallocating the array each time and worse, copying all elements in the list over to the new array. Especially when adding a large number of elements to a list (a very common operation), reallocation/copying would have to occur each time when a new element is added.

A more pragmatic way that is very common in practice is to double the size of the array as soon as it is full and to not change the container when an element is removed. This has the drawback that we often reserve more memory than we actually need (the container has more space than we need for the list) at the benefit of less frequent reallocations. Before considering the implementation of the list operations in detail, let us first theoretically investigate the number of element copies that this policy incurs. It is a simple example of what is called **amortized runtime analysis**.

Let us call the length of the array the **capacity** of the array list. So, assume that initially the array has capacity 1 and the list is empty, i.e. has length 0. (One could start with a higher capacity to optimize away some of the initial reallocations, but let us ignore that for the sake of simplicity.) Since we double the size of the array only when we want to add an element *and* the capacity equals the list size, the capacity will always be a power of two. Assume we are in this situation and the capacity (and the list length) is 2^n. If we now add one more element to the array list, we will allocate a new array of size 2^{n+1} and copy all 2^n elements over to the new array. So, in general, if the array capacity equals 2^m, we have copied $2^0 + 2^1 + \cdots + 2^{m-1} = 2^m - 1$ elements. However, if the capacity is 2^m, the array list contains at least $2^{m-1} + 1$ elements. So in total (*amortized* over all insertions) after inserting $2^{m-1} < M \leq 2^m$ elements to the array list, we have performed at most $2M$ copy operations.



(a) Extending the array by one element upon appending leads to a quadratic amount of copy operations.

(b) Doubling the capacity of the array leads to an amortized constant count of copy operations per element.

Figure 5.1.1 Evolution of array after appending five elements. Each row shows array after re-allocation. Arrows indicate copying of cells.

This analysis is *amortized* because it does not considers the complexity of a single append to the list, which can be expensive because we might copy the entire array, but a sequence of insertions over which the run time is amortized.

Basic Data Structure and Initialization. An array list consists of an array holding the list elements, represented by a pointer holding the address of the array container. Here, we will focus on integer elements and we discuss below (Genericity) how this can be generalized to arbitrary data types. Additionally, we need to memorize the current length of the list and the capacity of the array.

```c
 struct array_list_t {
    int* data;        // payload
    int size;         // number of elements in list
    int capacity;     // size of array;
};

typedef struct array_list_t array_list_t;
```

Open in Browser

Programmers can just declare variables of the above type. However, to function properly, they have to be initialized: The array has to be allocated and size and capacity have to be properly set. This is achieved by the following function. It takes a pointer to an (uninitialized) array list and an initial capacity and initializes the data structure accordingly.

```c
static void ensure_capacity(array_list_t* l, int
    new_capacity) {
    if (l->size == l->capacity) {
        assert (new_capacity > l->capacity);
        l->capacity = new_capacity;
        l->data = realloc(l->data, sizeof(l->data[0]) *
            l->capacity);
    }
}

array_list_t* array_list_new(array_list_t* l, int
    initial_capacity) {
    assert(initial_capacity > 0);
    l->size = l->capacity = 0;
    l->data = NULL;
    ensure_capacity(l, initial_capacity);
    return l;
}
```

Open in Browser

Remark 5.1.2 Allocation. You might ask yourself if it would be easier if `array_list_new` also allocated the memory for the `array_list_t` itself instead of requiring the user to give a pointer to it. This is pretty common in C programs because it gives the user more flexibility on where to allocate the memory for the variable. If `array_list_new` also allocated memory, it would have to use `malloc` which may not be what the user wants. Maybe they wanted a local or global variable instead.

Of course there also needs to be function that disposes all the allocated memory of an array list. Since this is just the array containing the list elements, this is a simple free. Note that it is still a good idea to provide a function for this because it relieves and prevents the programmer from studying the internals of the `array_list_t` data structure and *encapsulates* its inner working.

```c
void array_list_free(array_list_t* l) {
    free(l->data);
}
```

Open in Browser

Simple Functions. For a useful list implementation we also want functions to query the list size and get the element at the i-th position. These functions are very easy to implement and could be carried out by the user directly on the data structure. However, to ensure encapsulation, we provide functions nonetheless.

```c
int array_list_size(const array_list_t* l) {
    return l->size;
}

void* array_list_get(const array_list_t *l, int idx) {
    return idx < l->size ? l->data[idx] : NULL;
}

int array_list_find(array_list_t* l, int element) {
    for (int i = 0; i < l->size; i++) {
        if (l->data[i] == element) {
            return i;
        }
    }
    return array_list_size(l);
}
```

Open in Browser

Appending to an Array List. Appending implements the resizing of the array when the capacity is exhausted as discussed above. To this end, it uses the C standard library function `realloc` which takes a pointer to an existing container and a size in bytes and gives us back a pointer to a new container that has the given size and copies as much of the old data as will fit into the new container.

```c
void array_list_append(array_list_t* l, int element) {
    ensure_capacity(l, 2 * l->capacity);
    l->data[l->size++] = element;
    assert(l->size <= l->capacity);
}
```

Open in Browser

Removing an Element. Removing an element from a list involves copying "down" all subsequent elements with the element immediately following the one to be removed overwriting it. This is implemented by a simple for-loop. Note, that it is vital that the function also checks that the index is in bounds. Otherwise, undefined behavior (see Section 4.15) may occur when `idx` is out of bounds.

```c
int array_list_remove(array_list_t* l, int idx) {
    if (idx >= array_list_size(l))
        return 0;
    for (int j = idx; j+1 < l->size; j++) {
        l->data[j] = l->data[j + 1];
    }
    --l->size;
    return 1;
}
```

Open in Browser

Genericity. So far, our list elements were ints. However, one typically does not only want lists of ints but lists holding elements of an arbitrary type. We also say that the list (or the data structure in general) shall be **generic**. This means that the type of the elements is not known beforehand but the data structure is *parametric* in that data type, meaning that the data structure can be *instantiated* for arbitrary element types.

Genericity cannot be achieved very elegantly in C because C's type system does not provide a mechanism where types can be parameters to another type declaration like the declaration of the `array_list_t` struct. The standard way of attaining genericity in C is by using void pointers. By substituting `int element` by `void* element`, we indicate that the list element resides somewhere else in memory (not directly *inside* the array that holds the list). This makes the array list work with arbitrary element types.

However, the type system of C is too weak to provide helpful guarantees. Other programming languages like C++, Java, ML, and many others provide some sort of parametricity where the programmer can indicate that the list is actually a list of elements of some type T. Such type systems then ensure that the programmer can only add elements of type T to the list and, in turn guarantees that everything that is taken out of the list is also of type T. From a type system perspective, when using void pointers, we are essentially forgetting everything about the type so it is up to the programmer to cast the void pointer into appropriate element pointers.

Summary and Complexity Analysis. Array lists are very frequently used in practice. The amortized complexity of appending an element is constant as argued above. However, some append operations may take as many operations as there are elements in the list. Getting the i-th element can be done in a single operation by virtue of address arithmetic: We just index the array at index i. Removing and finding an element costs n copy operations in the worst case.

In terms of memory, array lists allocate twice as much memory as needed in the worst case.

Very important in practice is that the list elements lie contiguously in memory. Various hardware mechanisms (fetching cache lines, prefetching) are optimized for contiguously accessing arrays which makes array lists very fast in practice.

Exercises

1.

Modify the

```c
struct array_list_t {
    int* data;       // payload
    int size;        // number of elements in list
    int capacity;    // size of array;
};

typedef struct array_list_t array_list_t;

void ensure_capacity(array_list_t* l, int new_capacity)
    {
    if (l->size == l->capacity) {
        _assert (new_capacity > l->capacity, "The␣
            capacity␣should␣be␣monotone.");
        l->capacity = new_capacity;
        // l->data = relloc(l->data,sizeof(l->data[0])
            * l->capacity);
        // manual fix as realloc is not implemented
```

```
            int* newData = malloc(sizeof(l->data[0]) *
                l->capacity);
            for (int i = 0; i < l->size; i++) {
                newData[i]=l->data[i];
            }
            free(l->data);
            l->data=newData;
        }
}

array_list_t* array_list_new(array_list_t* l, int
    initial_capacity) {
    _assert(initial_capacity > 0, "The␣array␣should␣
        have␣a␣positive␣size.");
    l->size = l->capacity = 0;
    l->data = NULL;
    ensure_capacity(l, initial_capacity);
    return l;
}

void array_list_free(array_list_t* l) {
    free(l->data);
}

void array_list_append(array_list_t* l, int element) {
    ensure_capacity(l, 2 * l->capacity);
    l->data[l->size++] = element;
    _assert(l->size <= l->capacity, "The␣size␣should␣
        not␣exceed␣the␣capacity.");
}

int array_list_size(array_list_t* l) {
    return l->size;
}

int* array_list_get(array_list_t *l, int idx) {
    return idx < l->size ? &(l->data[idx]) : NULL;
}

int array_list_find(array_list_t* l, int element) {
    for (int i = 0; i < l->size; i++) {
        if (l->data[i] == element) {
            return i;
        }
    }
    return array_list_size(l);
}

int array_list_remove(array_list_t* l, int idx) {
    if (idx >= array_list_size(l))
        return 0;
    for (int j = idx; j+1 < l->size; j++) {
        l->data[j] = l->data[j + 1];
    }
    --l->size;
    return 1;
}


```

```c
int main() {
    array_list_t l;
    array_list_new(&l,5);
    printf("Size␣%d\n", array_list_size(&l));
    array_list_append(&l,1);
    array_list_append(&l,2);
    array_list_append(&l,3);
    printf("Size␣%d\n", array_list_size(&l));
    array_list_append(&l,32);
    array_list_append(&l,42);
    array_list_append(&l,125);
    printf("Size␣%d\n", array_list_size(&l));
    array_list_remove(&l,2);
    printf("Size␣%d\n", array_list_size(&l));
    printf("Pos␣of␣42␣%d\n", array_list_find(&l,42));
    printf("#3␣%d\n", *(array_list_get(&l,2)));

    array_list_free(&l);
    return 0;
}
```

Open in Browser

to provide genericity by using void pointers and write a small demo program that builds a list and uses the functions outlined above on it.

2.

The drawback of void pointer genericity is that we force the programmer to allocate the memory of the list elements elsewhere because the list only contains void pointers to the list elements. Come up with an implementation where the elements are stored inside the array. To this end, you have to change the implementation more drastically.

Hint. Figure out what the type of Introduce another field `element_size` that holds the size of an array element. Use this in the allocation function instead

5.1.2 Singly-Linked Lists

Array lists get you very far in practice and they are, for example, the implementation behind Python's lists, C++'s `std::vector`, and Java's `ArrayList`. As discussed above, modern hardware also goes to great lengths to optimize array processing which makes array lists very fast.

However, in some scenarios one does not want to pay the price of allocating potentially too much memory or the linear time complexity of removing elements from the list. In such cases, linked lists are practical.

A linked list uses pointers to create a chain of list elements. The list itself consists of nodes of which each contains a payload (the element) and, in case of a singly-linked list, a pointer to the next node in the list. If there is no next node, that next pointer is set to `NULL`. Each node is a container of its own that resides somewhere in memory. So there is no array that contains all nodes which makes it also impossible to access the i-th element in constant time because we need to chase the first $i - 1$ pointers to get get to the i-th node.

Basic Data Structure. A list node is typically a struct that consists of one field that carries the payload data; in our example a single `int` and another field that is a pointer to the next list node in the list. There may be variations to this scheme which we discuss below.

```c
 struct list_node_t {
    int element;
    struct list_node_t* next;
};
typedef struct list_node_t list_node_t;
```

Open in Browser

The pointer to the first element of the list is called the **root** of the linked list. This can be a `list_node_t` with an unused `element` field or a separate variable of type `list_node_t*`. The first alternative makes the following code more concise but wastes the space of the unused element field; a drawback that one can typically live with.

Inserting an Element behind Another Element. Inserting an element behind another element is simple for singly-linked lists and can be done in constant time. The only thing to do is to allocate a new list node, set its next pointer to the current first list node, and set the root to the freshly inserted element.

When using a "dummy" list node as the list root, the insert-behind operation can also be used to prepend an element to the list (insert it at the front). This makes using a list node for the root appealing because it saves an extra function that handles the special case for inserting at the front of the list.

```c
list_node_t* list_insert_behind(list_node_t* where, int
    element) {
    list_node_t* n = malloc(sizeof(*n));
    n->element   = element;
    n->next      = where->next;
    where->next = n;
    return n;
}
```

Open in Browser

Removing an Element behind Another Element. Removing an element from the list is equally simple: We just have to detach it from the list by redirecting the next pointer of the preceding element to the element following the one that is to be removed. Of course, we also need to free the memory allocated for the list node.

```c
int list_remove_behind(list_node_t* where) {
    list_node_t* to_be_removed = where->next;
    if (where->next == NULL)
        return 0;
    where->next = to_be_removed->next;
    free(to_be_removed);
    return 1;
}
```

Open in Browser

Finding an Element. Finding the list node for a certain element value (for example to check, if an element is in the list) takes linear time in worst case (in the worst case the element we are looking for is the last one in the list). The implementation traverses the list by "chasing" the next pointers until the end of the list is reached which is indicated by a next pointer of `NULL`. In C-style this can be written very compactly:

```c
list_node_t* list_search(list_node_t* root, int value) {
    list_node_t* n = root->next;
    while(n != NULL) {
        if (n->element == value)
            return n;
        n = n->next;
    }
    return NULL;
}
```

Open in Browser

Determining the length or getting the i-th element can be implemented in a similar style.

Genericity. With respect to genericity, the linked list implementation we discussed here suffers from similar problems as the array lists discussed in the previous section: There is no way to specify the list node data structure *parametric* to the data type of the payload. C forces us to give a concrete type like `int` which we used in the discussion above. Similar to array lists, void pointer genericity can be applied, making the elements void pointers to some data.

Another way that is also common in practice is to remove the `element` field from the list node struct and make this struct a member of the element data type like so:

```c
typedef struct {
  char* name;
  char* surname;
  struct {
    int day, month, year;
  } birthday;
  list_node_t list;
} person_t;
```

Open in Browser

The advantage is that the list functions can be generic, i.e. independent of the payload data structure. The drawback is that the `person_t` gets “contaminated” with the next pointer that is actually not part of a person.

Summary. Linked list have constant complexity for inserting and removing as opposed to array lists which have linear complexity for removing and *amortized* constant complexity for insertion. However, in contrast, the i-th element cannot be obtained in constant time in linked lists because, in general, there is no computation that takes *i* and returns the address of the i-th list node (like address arithmetic in the array list case).

Linked lists have a significant practical drawback compared to array lists: Since every list node is allocated individually, the list nodes are not guaranteed to sit contiguously in memory. This makes a lot of the machinery in modern microprocessors which are optimized for processing arrays useless. So searching and inserting are typically a lot faster on array lists than on linked lists although their theoretical complexity is identical.

In summary, the following table sums up the differences in complexity for array and singly-linked lists:

Operation	Array List	Linked List
Appending	amortized $O(1)$	$O(1)$
Inserting after i-th	$O(n)$	$O(1)$
Removing i-th	$O(n)$	$O(1)$
Getting pointer to i-th	$O(1)$	$O(n)$
Get size	$O(1)$	$O(n)$

One issue of *singly* linked lists is that inserting and removing *in front* of a given element is not directly possible because there are only next pointers and no pointers to previous elements. This can be remedied by using doubly-linked lists which we discuss in the next section.

5.1.3 Doubly-Linked Lists

As mentioned in the previous section, singly-linked lists have the problem that inserting before a given list node is complicated because there are no back pointers. Doubly-linked lists provide these.

```
 struct list_node_t {
    int element;
    struct list_node_t* prev;
    struct list_node_t* next;
};
typedef struct list_node_t list_node_t;

list_node_t* list_root_init(list_node_t* n) {
    n->next = n;
    n->prev = n;
}
```

Open in Browser

In the same way as with singly-linked lists, using a dummy list node for the list root is advisable because it keeps the algorithms simpler. In contrast however, we organize the list as a *ring* such that if the prev pointer of an element is the root that element is the first element in the list and analogously if the next pointer of an element is the root, that element is the last element. Initially, the root's next and prev pointer point to the root itself. This way, inserting and removing do not need special case handling for the first and last element of the list.

We will briefly discuss insertion and removal. Searching and the other simple algorithms are similar to singly-linked lists.

Insertion. Inserting into a doubly-linked list involves entering a freshly allocated list node into the link structure. Note that it doesn't really matter if we implement appending (inserting behind a node) or prepending (inserting before a node) because prepending to a node becomes appending to its predecessor which always always exists if we use a dummy element as a root because the doubly-linked list then effectively becomes a ring.

```
list_node_t* list_append(list_node_t* where, int element) {
    list_node_t* n = malloc(sizeof(*n));
    n->element    = element;
    n->next       = where->next;
    n->prev       = where;
    where->next   = n;
    n->next->prev = n;
    return n;
}
```

Open in Browser

Removal. When removing a node, we specify the node directly and dismount it from the list by setting the next pointer of its predecessor to its next node and the previous pointer of its next element to its previous element:

```
void list_remove(list_node_t* n) {
    n->next->prev = n->prev;
    n->prev->next = n->next;
    free(n);
}
```

Open in Browser

Checkpoint 5.1.3 Doubly Linked Lists. Implement insertion and removal for doubly linked lists without using a dummy list element. Make sure that:

- `prev` of the first element and `next` of the last element are both always `NULL` (in contrast to the implementation *as a ring* in the lecture notes).
- `head` always points to the first element of the list and `tail` to the last unless the list is empty, in that case both should be `NULL` .

Use the following struct and implement the five declared functions.

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

struct list_node_t {
    int element;
    struct list_node_t* prev;
    struct list_node_t* next;
};
typedef struct list_node_t list_node_t;

list_node_t* list_root_init(list_node_t** head,
    list_node_t** tail) {
    *head = NULL;
    *tail = NULL;
}
```

Open in Browser

Compare your implementation with the one in the lecture notes:

- Why do you need the arguments head and `tail` in contrast to the implementation as a ring, where the root element is not passed to in the functions for insertion and removal?
- Why do you need a separate function to prepend an element, but don't need it in the other implementation?
- Why do you need functions for prepending (appending) at head (tail)?

Solution.

```c
        struct list_node_t {
            int element;
            struct list_node_t* prev;
            struct list_node_t* next;
        };
        typedef struct list_node_t list_node_t;

        list_node_t* list_root_init(list_node_t** head,
            list_node_t** tail) {
            *head = NULL;
            *tail = NULL;
        }

        list_node_t* list_prepend_at_head(list_node_t**
            head, list_node_t** tail,
                                        int element) {
```

Open in Browser

- The dummy list node in the other implementation is the analog to the `head` and `tail` pointers in this one. But there the first and last list element have pointers to the dummy node, thus it is accessible from within the list. This is not the case when the first and last elements point to `NULL`, so we have to pass them as double pointers to be able to change them appropriately.
- In the implementation as a ring we can always just append it to the previous list node instead of prepending (or vice versa: instead of appending we could prepend to the next node). Here, the previous (or next) node does not necessarily exist, so there has to be some case distinction.
- Appending to the dummy node is equivalent to `list_prepend_at_head` and prepending to the dummy node is equivalent to `list_append_at_tail`. Without (one of) these functions it would not be possible to add the first element to an empty list.

5.2 Trees

Trees are an important data structure to represent data hierarchically. Trees generalize lists in that one node can have multiple (possibly no) successors, called **children**. However, each node has exactly one **parent** node, except for the **root** of the tree which has no parent. Trees are used in a large variety of contexts. Most notably perhaps are they used to implement sets and maps.

5.2.1 General Notions and Algorithms

For each tree, there is a **directed graph** in which the nodes of the graph are the tree nodes and there is an edge in the graph from node a to a node b if b is a child of a. A node in a tree that has no children is called a **leaf**. A node a is a **descendant** of another node b if there is a **path** of edges from a to b in the tree's graph. A tree is a **binary tree** if each node has at most two children. A tree is **labelled** if each node carries additional information, such as a number or a string for example. That label is often also referred to as a **key**. It is very common to demand that the children of a node are *ordered* so that it actually matters if a child is left or a right child in a binary tree, for example. So in fact, when talking about trees, we most often talk about *ordered labelled trees*.

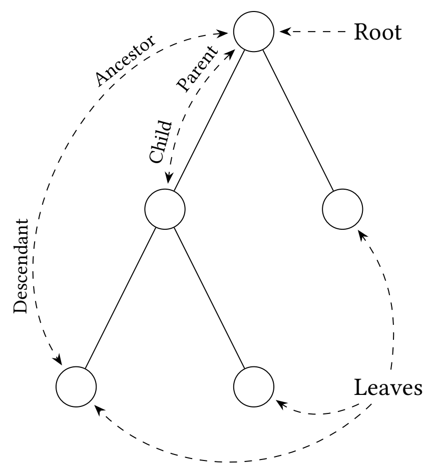


Figure 5.2.1 A binary tree with three leaf nodes.

Implementation. The most straightforward implementation of a tree is to have a struct containing fields for the label(s) and pointers to the children of a node. It is common that the number of possible children is fixed, e.g. in a binary tree there are at most two children. In that case, the child pointers can be either individual fields in that struct (like a `left` and `right` field in the case of a binary tree) or an array of constant size. A child pointer is set to `NULL` if the node does not have any children. If each child pointer equals `NULL` then the node is a leaf.

```
 struct binary_tree_t {
  int key;
  struct binary_tree_t *left;
  struct binary_tree_t *right;
};

struct tree_t {
  int key;
  struct tree_t *children[10];
};
```

Open in Browser

Remark 5.2.2 Sets as Maps. A set can be seen as a map where

- the domain of the map is the set
- the codomain of the map is a singleton set (a set with one member)
- the map maps each set element to the singleton

So, often one implements the more general map case where each tree node has a *key* and a *value*. The key acts as the label of the node and the value is the value the key is mapped to. When one is just interested in a set, the value is just set to a default value (like `0` or `NULL` for example).

Height, Depth, Size. The **height** of a node is the length of the longest path from that node to a leaf. The height of the tree is the height of root. Computing the height can, like many tree algorithms, be done nicely in a recursive fashion.

```
int binary_tree_height(binary_tree_t* n) {
  if (n)
```

```
    return 1 + max(binary_tree_height(n->left),
                   binary_tree_height(n->right));
  else
    return 0;
}
```

Open in Browser

The **depth** of a node is its distance (in edges) from the root.

The **size** of the tree is the number of its nodes.

5.2.2 Binary Search Trees

A binary search tree (BST) is an ordered labelled binary tree that requires the existence of a total order on the nodes' keys. For example, the labels could be integers and the total order the standard less-than comparison.

Definition 5.2.3 Binary Search Tree Property. A tree is a binary search tree, if for each node n with label ℓ it holds that

1. the key of each node in the *left* subtree of n is less than ℓ
2. ℓ is less than the key of each node in the *right* subtree of n

◊

By this definition there exist no two nodes in a binary search tree that have the same key.

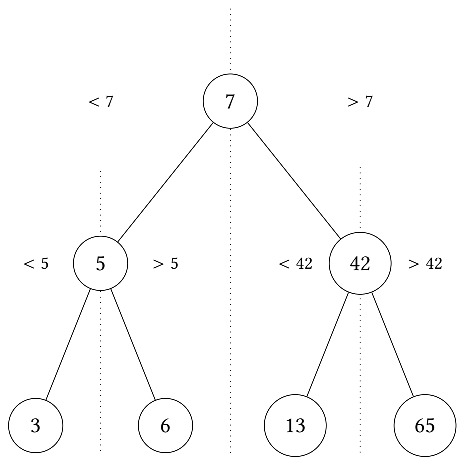


Figure 5.2.4 A binary search tree. Every node left of the root is smaller than seven. Every node on the right is larger.

Searching. Searching for a key in a binary search tree can be nicely done using a tail-recursive algorithm. First, we compare the key k that is searched with the key of the root. If they are not equal, the search tree property tells us if we must look for the key in the left *or* the right subtree. If k is smaller than the root's key, all nodes in right subtree have greater keys than k, so if k is in the key, it will be in the left subtree (and vice versa).

This procedure will take at most as many steps as the tree is high. If the binary search tree is **balanced**, the height of the tree is bounded by the binary logarithm of its size. A node is *balanced*, if the height of its left subtree differs by at most one from the height of its right subtree. A binary tree is balances if all its nodes are balanced.

Checkpoint 5.2.5 Balanced Binary Trees. Prove: For a balanced tree of height h

and size n, holds:

$$h \leq 2\log_2 n.$$

Solution. We first prove the dual formulation of the above statement:

$$n \geq 2^{h/2}.$$

To do this, we introduce the minimal size of a balanced tree of height h as $s_{min}(h)$.

$h = 0$: The tree is a single node.

$$s_{min}(h) := 1.$$

$h = 1$: The tree is either a root with a single child or a root node with two children. The first case has less nodes, so we define:

$$s_{min}(h) := 2.$$

$h-1, h \longrightarrow h+1$: By the definition of height, one subtree has height $h-1$. Using proof by contradiction, we can show that the other subtree has height $h-2$ as the size s_{min} would not be minimal otherwise. By recursion, we obtain the minimal sized trees for $h-1$ and $h-2$.

$$s_{min}(h+1) := s_{min}(h) + s_{min}(h-1) + 1.$$

By definition of s_{min}, we have for every tree of size n and height h:

$$n \geq s_{min}(h).$$

We show $s_{min} \geq 2^{h/2}$ to obtain $n \geq 2^{h/2}$ by transitivity.

By complete induction on h:

$h = 0$:

$$s_{min}(h) = s_{min}(0) = 1 \geq 1 = 2^0 = 2^{h/2}.$$

$h = 1$:

$$s_{min}(h) = s_{min}(1) = 2 \geq \sqrt{2} = 2^{0.5} = 2^{h/2}.$$

$h = 2$:

$$\begin{aligned} s_{min}(h) &= s_{min}(h-1) + s_{min}(h-2) + 1 \\ &= s_{min}(1) + s_{min}(0) + 1 \\ &= 2 + 1 = 3 \geq 2 = 2^1 \\ &= 2^{h/2} \end{aligned}$$

$h-1, h \longrightarrow h+1$: Induction hypothesis: $s_{min}(h-1) \geq 2^{(h-1)/2} = 2^{h/2-1}$

$$\begin{aligned} s_{min}(h+1) &= s_{min}(h) + s_{min}(h-1) + 1 \\ &= (s_{min}(h-1) + s_{min}(h-2) + 1) + s_{min}(h-1) + 1 \\ &= 2s_{min}(h-1) + s_{min}(h-2) + 2 \\ &\geq 2s_{min}(h-1) \\ &\overset{IH}{\geq} 2 \cdot 2^{h/2-1} \\ &= 2^{h/2} \end{aligned}$$

Lastly, we obtain from $n \geq 2^{h/2}$ the inequality:

$$h \leq 2\log_2(n).$$

```c
binary_tree_t* binary_tree_search(binary_tree_t* root, int
    k) {
  if (!root || root->key == k)
    return root;
  else if (k < root->key)
    return binary_tree_search(root->left, k);
  else
    return binary_tree_search(root->right, k);
}
```

Open in Browser

Insertion. One simple way to insert a key into binary search tree while maintaining the search tree property is to insert new nodes as leaves. to this end, we need to find the suitable current leaf under which the new node has to be inserted. This is very similar to searching for the node but instead of returning NULL when the key was not found, we return the leaf at which the search ended. Under this leaf the new node will be added to the tree. Either as its left child if the key to be inserted is smaller than the key of the leaf or as its right child otherwise.

```c
binary_tree_t* binary_tree_insert(binary_tree_t** root, int
    k) {
  binary_tree_t* y = NULL;
  binary_tree_t* x = *root;

  while (x) {
    // y now points to the parent of x
    y = x;
    if (k == x->key)
      return x;
    else if (k < x->key)
      x = x->left;
    else
      x = x->right;
  }

  // The key is not in the tree, so allocate a new tree node
  binary_tree_t* new_node = malloc(sizeof(*new_node));
  new_node->left = NULL;
  new_node->right = NULL;
  new_node->key = k;

  if (y) {
    // y points to a leaf
    if (k < y->key)
      y->left = new_node;
    else
      y->right = new_node;
  }
  else {
    // if y is NULL, then the tree is empty
    *root = new_node;
  }

  return new_node;
}
```

Open in Browser

Note that insertion into a *balanced* binary search tree can destroy the balanced-ness property. There exist various extensions to binary search trees (such as AVL trees[9] or red-black trees[10]) that preserve balanced-ness upon insertion and deletion by modifying the tree accordingly. These are out of scope of this book and we refer to a standard algorithms and data structures text (e.g. [4]) for more detailed information.

5.2.3 Tries

A trie (sometimes also called **prefix tree**), is a search tree specifically designed to implement maps (and therefore sets, see Remark 5.2.2) where the keys are *words (sequences)* of 'items' that are drawn from a given set Σ. Here, a word could for example be a string (a sequence of characters) or also a number (as a sequence of digits, see Chapter 1).

One possibility to implement such a map would be to use the words directly as keys in a binary search tree. However, comparing two sequences boils down to comparing individual elements in practice. For example, consider comparing "Helmet" against "Hello": We need to compare three characters (H, e, l) before we can discriminate both words because both words share the common prefix "Hel". So comparing two sequences takes in the worst case as long as the shorter of both words. Since we might need to visit several nodes of the search tree until we either find the sequence in question or verified that it is not contained in the map, the overall worst-case runtime is the maximum depth of the BST *times* the length of the sequence we are looking for.

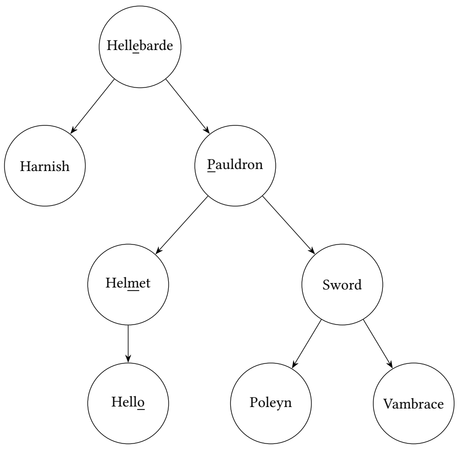


Figure 5.2.6 The search of "Hello" in a binary search tree. For every comparison, the deciding letter is underlined.

Tries reduce this worst-case runtime to the length of the sequence that is searched for by using a different kind of tree structure. The idea is that for each word in the map (set), there is exactly one path from the trie's root to a leaf and vice versa. So each node in a trie has at most $|\Sigma|$ children and each the edge stands for one occurrence of an 'item' in a word. More precisely, consider a path π from the trie's root to a leaf and let n_i be the i-th node on this path. If n_i is not a leaf and n_{i+1} is the j-th child of n_i, then the i-th item in the word that the path corresponds to, is j.

[9] `wikipedia.org`
[10] `wikipedia.org`

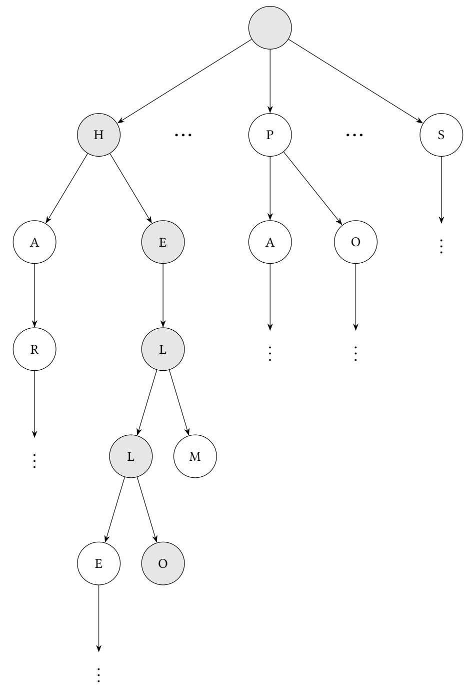


Figure 5.2.7 The tree from above but as trie. The path π is marked.

To illustrate inserting and searching in a trie, we will use a trie that implements a map from ints to some other data type T. So, a *word* now is the sequence of bits in an int. Therefore, each node in the trie will have at most two children (each corresponding to the digits 0 and 1). An int that is in the map the trie implements is therefore represented by a path that is as long as the number of bits of an int (`sizeof(int) * 8`).

Searching. When searching for a key in the map, we iteratively visit nodes in a path that starts at the root. In each step, we use the next bit in the key to identify the next child node to visit: If the next bit is 0, we pursue the child 0, if it is 1, we pursue the child 1. If the identified child node exists in the trie, we visit it, if not (indicated by a `NULL` pointer), the key is not in the trie and the search returns false. If we succeed to visit as many nodes as the key has bits, the key is in the map.

```c
int_trie_t* int_trie_search(int_trie_t* root, int key) {
  int n = sizeof(key) * 8;
  while (n > 1) {
    _Assert (root != NULL, "A correctly initilized tree can 
        not be NULL.");
    int_trie_t* next = root->next[key & 1];
    if (! next)
        return NULL;
    root = next;
    key = key >> 1;
    n--;
  }

  _Assert(n == 1, "We should be at the leaves in the end.");
  return root;
}
```

Open in Browser

Insertion. Insertion is very similar to searching. The only difference is that if we want to visit a child that is not in the trie, we do not abort and return false but create a new node and insert it.

```
int_trie_t* int_trie_insert(int_trie_t* root, int key, T
    value) {
  int n = sizeof(key) * 8; // number of bits in key
  while (n > 0) {
    _Assert(root != NULL, "A␣correctly␣initilized␣tree␣can␣
        not␣be␣NULL.");
    int_trie_t* next = root->next[key & 1];
    if (! next) {
        next = int_trie_create();
        root->next[key & 1] = next;
    }
    root = next;
    key = key >> 1;
    n--;
  }
  root->value = value;
  return root;
}
```

Open in Browser

5.3 Hashing

We have seen that search trees can implement maps (and therefore sets). Another data structure for sets (and maps) that is often used in practice are hash tables. The basic idea is that we define a **hash function** h from the set of keys K to `int` to assign each possible key a **bucket** in a **hash table** which is just another way of saying 'an index in an array'. If h is *injective* (i.e. no two keys get assigned the same hash value) *and* the array is large enough, each key can get its own individual bucket. We then also say that there are no **collisions**. Checking if a key k is in the set then reduces to evaluating $h(k)$ and checking if the array is occupied at $h(k)$. Being occupied could for instance mean that at $h(k)$ is a value that is associated with k in case of a map or just a 1 to indicate that k is in the set. If evaluating $h(k)$ does not depend on the size of k we can check membership (and also add and remove) in constant time.

However, in reality, neither is h injective nor is the array large enough. h is rarely injective because the cardinality of the key set is typically much higher that the one of `int`. For example, in the case where the keys are strings, there are strictly more strings that ints. Even if h was injective, since `int` comprises 2^{32} values, we would need an array of size 2^{32} to safely avoid all collisions.

So if we have to give up on injectivity anyways, we can also require that the hash function always yields a value within size of our array. In fact, for each hash function h we can define a new function $h_m(k) = h(k) \mathbin{\%} m$ where m is the length of the array. This function always yields a value between 0 and m.

In the following, we look at two different techniques for dealing with collisions and still keep the run time of membership tests "low" in practice.

5.3.1 Separate Chaining

The first straightforward idea is to resolve collisions using a list for each bucket. To this end, we are not placing the values (which are associated with the keys) directly into the hash table but in fact make the array an array of lists. The idea is that all elements that are hashed to the same bucket are chained in the collision list for that bucket.

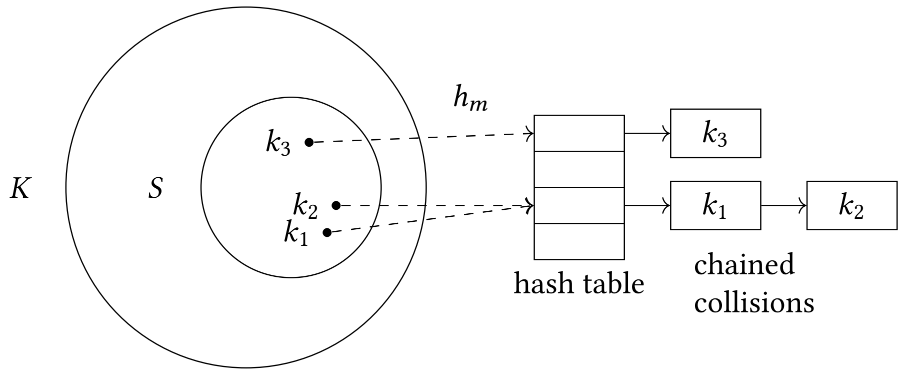


Figure 5.3.1 Chained collisions in a hash table. The hash table represents the set of keys $K \supseteq S = \{k_1, k_2, k_3\}$. Keys k_1 and k_2 collide under h_m.

The following code shows a sample implementation of a hash table with separate chaining in C. The function `hash_chain_search` searches for a given key. It returns 1 if the key is in the table and 0 otherwise. In any case, it stores into `insert` the address of an entry pointer. If a new entry must be added to the hash table (when inserting an element), this pointer needs to be set to the freshly added entry.

```
typedef int (*eq_t)(void* p, void* q);
typedef int (*hash_fct_t)(void* key);

typedef struct entry_t {
    void* key;
    struct entry_t* next;
} entry_t;

typedef struct {
    unsigned size;           // # of keys in the table
    unsigned table_length;   // length of the hash table
    float max_load_factor;   // the maximal load factor
    hash_fct_t h;            // the hash function.
    entry_t** table;         // an array of singly-linked
        entry lists
} chaining_hashtable_t;

/// This function searches for a key
/// in a hash tab with separate chaining.
/// @param ht The hash table
/// @param key The key to search for
/// @param hash The hash function value for key
/// @param eq A function to compare two keys for equality
/// @param insert The address of a pointer to an entry_t
    pointer.
///                There, we will put the address of the
    pointer
///                where the new collision chain entry
    should be
///                appended in the case where we insert a
    new key.
int hash_chain_search(chaining_hashtable_t* ht, void* key,
                      eq_t equals, entry_t*** insert) {
    unsigned bucket = ht->h(key) % ht->table_length;  //
        compute h_m
    *insert = ht->table + bucket;
    for (entry_t* e = ht->table[bucket]; e != NULL; e =
        e->next) {
        if (equals(e->key, key))
            return 1;
        *insert = &e->next;
    }
    return 0;
}
```

Open in Browser

Listing 5.3.2 Searching in a hash table with separate chaining. The insert parameter will be set to the address of a pointer to an entry. This pointer can be used by the insertion function to chain-in a new entry.

```
/// Insert into a hash table with separate chaining.
int hash_chain_insert(chaining_hashtable_t* ht, void* key,
    eq_t equals) {
    entry_t** insert;
    int found = hash_chain_search(ht, key, equals, &insert);
    if (found) {
        // the hash table did not change
        return 0;
    }
    // The search function sets insert to the address of
    // the last next pointer in the collision chain
    assert (*insert == NULL);
    // here, we know that the element is not in the table
    // so we allocate a new list entry
    entry_t* e = malloc(sizeof(*e));
    // initialize the entry
    e->next = NULL;
    e->key = key;
    // and append it to the collision chain
    *insert = e;
    ++ht->size;
    return 1;
}
```

Open in Browser

Listing 5.3.3 Inserting into a hash table with separate chaining.

5.3.2 The Load Factor

Consider the situation that we were to add n keys into a hash table with m entries. Under the optimistic assumption that the hash function $h(k)\ \%\ m$ distributed the n keys *evenly* across the hash table, each collision chain has length $\alpha = n/m$. This number α is called the load factor of the hash table. The operations add, remove, search (check for containment) can, under this assumption, be carried out in $O(1+\alpha)$ operations.

Of course the assumption that the hash function distributes the data evenly across the table is in general hard to prove. Especially, for a given hash function, it depends on the concrete set of keys to be inserted how evenly the keys will be scattered over the table. And of course, because the hash function is in general not injective, by the **pigeonhole principle**[11], for each bucket there exists a “worst-case” of keys that all map to this bucket.

In practice, one is interested in keeping $\alpha < 1$ which means that, in the ideal case, no collision contains more than one key. To achieve this, on each insertion operation the load factor is checked and if it passes a certain threshold (0.75 is a common value) the hash table is resized. Note that when the hash table is resized, h_m also changes and all elements in have to be *rehashed* into the new table. Doubling the size of the table yields an amortized constant number of rehashing operations for the same argument made in Subsection 5.1.1.

[11]The pigeonhole principle says that if n keys are mapped to m entries with $n \geq m$, then at least one entry must contain more than one key.

```c
int hash_chain_insert_resize(chaining_hashtable_t* ht,
    void* key, eq_t eq) {
    if (ht->size > ht->max_load_factor * ht->table_length) {
        ht->table_length *= 2;
        assert (ht->size < ht->max_load_factor *
            ht->table_length);
        entry_t** old_table = ht->table;
        // allocate new table and initialize to 0
        ht->table = calloc(ht->table_length,
            sizeof(ht->table[0]));
        // go over old hash table and add each element to
            new table
        rehash_table(ht, old_table);
        // free old table
        free(old_table);
    }

    return hash_chain_insert(ht, key, eq);
}
```

Open in Browser

Listing 5.3.4 Inserting into a hash table with separate chaining with resizing of the hash table if the load factor threshold is exceeded. The function `rehash_table` traverses all collision chains of the old table and inserts the elements into the new table.

5.3.3 Hash Tables and Mutability

When hashing mutable data (cf. Section 8.6) one has to be extremely cautious. Typically, the value of the hash function depends on the fields of the struct/class. Changing the values of the fields while a certain object is referenced from a hash table may suddenly make the hash value with which it was hashed in and therefore the hash table index invalid so that it cannot be found anymore.

Consider the following example:

```c
typedef struct {
  int numerator, denominator;
} fraction_t;

int hash_fraction(fraction_t* f) {
  int d = f->d;
  int n = f->n;
  return (d + n) * (d + n + 1) / 2 + n;
}
```

Open in Browser

Here, the hash value for a data type that represents fractions is computed using the fraction's numerator and denominator. For example, the fraction 2/7 has the hash value 52. Assume that such a fraction is inserted to a hash table with a table length longer than 52. Then, it will be appended to the collision chain of bucket 52. If, after it has been inserted, the fraction is changed to, say, 2/9 the hash value changes and the fraction is in the collision chain of the wrong bucket. Searching for 2/7 in that hash table will consequently yield a negative result.

5.3.4 Open Addressing

Chaining collisions in dedicated linked lists has the disadvantage that the linked lists incur additional memory overhead for the list elements and that linked lists have more or less random memory access patterns (see the discussion at the end of Section 5.1). Therefore, one often uses **open addressing** (also known as **probing**) where the collision chain is stored in the hash table itself.

To this end, we define a new function

$$h' : K \times \mathbb{N} \longrightarrow \{0, \ldots, m-1\}$$

which, in addition to the key, also takes a number i that serves as the position of key in its collision chain, and yields the table index where the i-th element of k's collision chain shall be stored. When inserting a new key k into the hash table, we compute $h'(k, 0)$ and check if the hash table is occupied at this position. If not, we enter k there. If yes, we compute $h'(k, 1)$ and check again, and so on.

```c
typedef struct {
    unsigned size;
    unsigned table_length;
    float max_load_factor;
    hash_fct_t h;
    void** table;
} open_addr_hashtable_t;

int open_addr_hashtable_search(open_addr_hashtable_t* ht,
                               void *key, eq_t equals) {
    for (unsigned i = 0; i < ht->table_length; i++) {
        // call probing function to compute bucket
        // of collision chain entry
        unsigned bucket = h_prime(ht->h(key), i);
        void* k = ht->table[bucket];
        if (k == NULL || equals(k, key))
            return bucket;
    }
```

Open in Browser

Listing 5.3.5 Inserting into a hash table with open addressing.

The function `probing_hashtable_search` returns the table index of a key equal to key if such a key is in the table. If there is no such key, it returns the index in the hash table where key can be inserted. If all entries $h'(k, 0), h'(k, 1), \ldots, h'(k, m-1)$ are occupied it returns the error code -1. The pressing question now is if the table really is full in this case, i.e. if h' is *surjective* with respect to its second parameter for each key. This certainly depends on how we define h'. In the following, we will discuss two different ways of defining h' for which hold that

$$\{h'(k, 0), h'(k, 1), \ldots, h'(k, m-1)\} = \{0, \ldots, m-1\}$$

i.e. that whenever the search function returns -1 the table is indeed full.

Linear Probing. The simplest approach to surjective probing is **linear probing**:

$$h'(k, i) := (h(k) + i) \ \% \ m$$

Linear probing trivially guarantees that each bucket will be searched for k, i.e. if there is a free bucket, linear probing will find it. A disadvantage of linear probing is that in practice sometimes clusters will form at certain entries which can impede performance of the search operation (see Listing 5.3.5): The larger the clusters the further away from $h'(k, 0)$ will be the actual bucket for a key k and the more the run time for searching increases. An advantage however is the linear access pattern in which the table is traversed. This may lead to better cache locality and favor several microarchitectural optimizations modern processors make.

Quadratic Probing. **Quadratic probing** was introduced to solve the problem of cluster formation of linear probing. To this end, one introduces a quadratic term to "push" higher collision chain entries further away:

$$h'(k, i) \coloneqq (h(k) + ci + di^2) \ \% \ m$$

Now the question is can c and d chosen such that h' is surjective on the table for each key. (Again, if it is not, this implies that we would not be able to add a key even if the table was not full.) The following theorem establishes that by setting $c = d = 1/2$ makes h' *injective* on $\{0, \dots, s^k - 1\}$ for any $k \geq 0$.

Theorem 5.3.6 *$h'_a(i) \coloneqq (a + i(i+1)/2) \ \% \ 2^k$ is injective on $\{0, \dots, 2^k - 1\}$.*

Proof. By contradiction: Assume there is $h_a(i) = h_a(j)$ for two numbers $i \neq j$ in the set $\{0, \dots, 2^k - 1\}$. Then,

$$a + i(i+1)/2 \equiv a + j(j+1)/2 \mod 2^k$$

and there is a $q \in \mathbb{Z}$ such that

$$\begin{aligned} & i(i+1)/2 - j(j+1)/2 = q \cdot 2^k \\ \iff \quad & i(i+1) - j(i+1) = q \cdot 2^{k+1} \\ \iff \quad & (i+j+1)(i-j) = q \cdot 2^{k+1} \end{aligned}$$

Now, the first observation is that for $i \neq j$ the product $(i+j+1)(i-j)$ is also unequal to zero, so we have $q \neq 0$. Second, for all i, j holds that if $(i-j)$ is odd, then $(i+j+1)$ is even. This means that exactly one of both factors is a multiple of 2^{k+1}. But since i and j are chosen from $\{0, \dots, 2^k - 1\}$, both $(i+j+1)$ and $(i-j)$ are less than 2^{k+1} which is a contradiction. ■

Corollary 5.3.7 *$h'_a(i) \coloneqq (a + i(i+1)/2) \ 2^k$ is surjective on $\{0, \dots, 2^k - 1\}$.*

Proof. Since the domain of h'_a is equal to the codomain, Theorem 5.3.6 implies that h'_a is a bijection and hence surjective. ■

5.3.5 A Comprehensive Example

Let us contrast collision resolution by chaining, linear and quadratic probing. We successively add the following strings into the hash tables

```
Hash Assertion Heap Hoare Binary
```

For the sake of simplicity, we use the index of first letter in the alphabet as the hash value of the string, i.e. $h('\mathtt{A}') = 0, h('\mathtt{B}') = 1$, etc. Note that in practice this is not a good hash function because it throws all strings that start with the same letter into the same bucket. Better hash functions would consider at least multiple characters of the string and maybe also weight them differently to differentiate permutations.

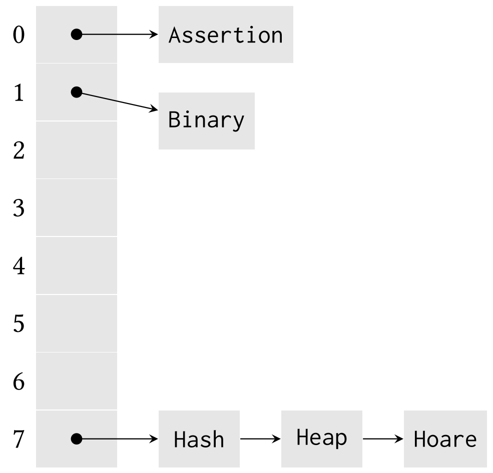


Figure 5.3.8 Collision resolution with separate chaining.

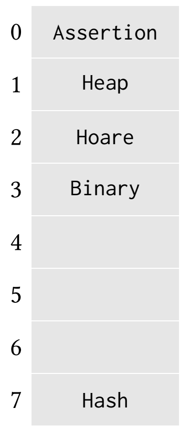


Figure 5.3.9 Collision resolution with open addressing linear probing.

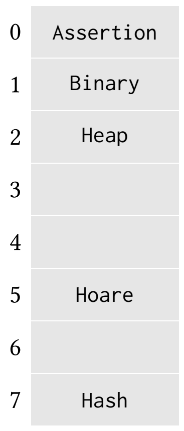


Figure 5.3.10 Collision resolution with open addressing and quadratic probing.

5.3.6 Deleting from a Hash Table with Open Addressing

An interesting situation arises when deleting an element from an open addressing hash table. Let us consider the example in Figure 5.3.9 and assume that we want to remove the key 'Heap' after having inserted all the keys. If we just remove "Heap" from the table and set the table entry to `NULL`, there will be a "hole" in the linear probing sequence for entry `Hoare`. So if we search for `Hoare` after having removed `Heap` this way, we would not find it anymore because the search function (Listing 5.3.5) would interpret the collision chain for `Hoare` to be terminated at index 1.

This problem can be solved by not setting the table entry of the removed key to `NULL` but to a special value, sometimes called **tombstone** to indicate that there *was* an entry. When searching for insertion points, tombstones count as empty entries, when searching for containment checking, they count as occupied entries.

5.3.7 Summary

We have merely scratched the surface of hashing and only explained some of the basic concepts.

Hashing is widely used in practice because it is very fast, typically faster than search trees. Especially open addressing hash tables, although on first sight less intuitive as separate chaining hash tables, are considered to perform very well mostly because they have very good memory access behavior due to their locality: everything is concentrated in one array. One may want also want to directly place the hashed key-value pairs into the table and not operate with `void*` pointers as we did in our examples for the sake of simplicity. In practice that typically depends on the size of the key-value pair.

We have said little about what makes a good hash function. This is a wide field and often also subject to empirical evaluations. Intuitively, a good hash function should scatter the data evenly across the hash table but it is not straightforward to say something substantial on specific hash functions.

We have also omitted a solid formal derivation of worst-case run time bounds because this is in general not possible without making further assumptions. One very prominent formal approach is *universal hashing* where one draws *hash functions* randomly and is able to prove run time bounds for specific hash functions independent of the data.

Although hashing is not easy to come by theoretically, it is, although there exit pathological worst-case scenarios, very popular in practice because it is very fast in the average case. We refer the interested reader to a recent extensive experimental study that explores many different dimensions in the hashing design space [15].

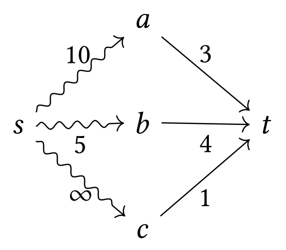
5.4 Dynamic Programming

A very common technique in algorithm design is **divide and conquer**. Hereby, the problem to solve is split into several sub-problems that are solved individually. After they have been solved, their solutions are combined to a solution into the original problem. A prominent example for a divide-and-conquer algorithm is **merge sort**. Merge sort splits the array in half, applies itself recursively to each of the halves and recombines the two sorted halves into a fully sorted array using a linear-time algorithm.

In some problems we want to solve, there is however the situation that some of the sub-problems appear several times, i.e. we have **overlapping sub-problems**. A straightforward recursive implementation would recompute their solution over and over again. To avoid unnecessary recomputation, we can use *memory* to **memoize** the result of the first solution of such a subproblem and reuse the memoized value in any further situation that solution of the sub-problem is required.

Dynamic programming is a programming technique to solve certain mathematical optimization problems using memoization. An optimization problem is a problem where there are multiple *valid* solutions of which each has a certain cost. One is then interested in finding the *best* solution with respect to that cost. For example, in route planning there may be several routes that get you from s to t. However each route has a certain length and so there will be shorter and longer routes. Hence, routing is an optimization problem because one is interested in finding the *shortest route*.

Dynamic programming is applicable if the optimization problem has so-called **optimal substructure**. This means that problem can be, in a way similar to divide-and-conquer, partitioned into sub-problems *and* the *optimal* solution of the original problem can be efficiently obtained from the *optimal* solutions of the sub-problems. Finding the shortest route/path from a node s in a directed graph to another node t is an optimization problem that possesses optimal substructure: Having found the shortest path from s to each predecessor of t allows us to quickly identify the shortest path to t itself by locally considering all incoming edges of t (see Figure 5.4.1).



$$sp(s, v) = \min\{10 + 3, 5 + 4, \infty + 1\} = 9$$

Figure 5.4.1 We can compute the length of the shortest path from s to t from the lengths of the shortest paths to t's predecessors.

We will first discuss two examples that have overlapping sub-problems to investigate how we can replace straightforward recursion using memoization. Then, we will consider some optimization problems that have optimal substructure and devise dynamic programming algorithms to solve them.

5.4.1 Computing Fibonacci Numbers

We start with a simple example that has overlapping sub-problems: Computing Fibonacci numbers. These are recursively defined as follows:

$$F_0 := 0 \qquad F_1 := 1 \qquad F_n := F_{n-1} + F_{n-2} \text{ for } n > 1$$

A first straightforward recursive implementation looks as follows:

```
 unsigned fib(unsigned n) {
  if (n <= 1)
    return n;
  else
    return fib(n - 1) + fib(n - 2);
}
```

Open in Browser

Let's consider the call `fib(6)`. Figure 5.4.2 shows the *call tree* of `fib(6)`.

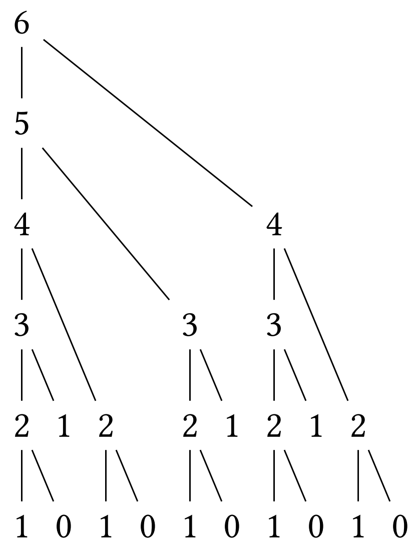


Figure 5.4.2 The call tree of `fib(6)`. Each node corresponds to an argument to a call to `fib`. An edge indicates that the call with the upper argument calls `fib` with the lower argument.

The number of calls `fib(n)` creates is F_n. For example, $F_{45} = 1,836,311,903$. So, assuming that one call takes 10 nanoseconds, `fib(45)` will take 45 seconds to compute. This can be drastically accelerated because the simple recursive scheme computes the same calls multiple times. For example, `fib(4)` is computed twice. By memorizing the result of the first call, all subsequent calls with the same argument can be avoided.

```
unsigned fib_memoize_c(unsigned n, int* table) {
  if (n <= 1)
    return n;
  else if (table[n] == 0)
    table[n] = fib_memoize_c(n - 1, table)
             + fib_memoize_c(n - 2, table);
  return table[n];
}

unsigned fib_memoize(unsigned n) {
  unsigned table[n+1];
  memset(table, 0, sizeof(table));
  return fib_memoize_c(n, table);
}
```

Open in Browser

This version uses a table to memoize all results of recursive calls and checks if the result of a call has already been computed in the past. If so, that value is read from the table and not recomputed. However, when looking at the computation of the Fibonacci numbers more closely, one observes that we only need the last two values, so memoizing everything is not necessary. We can achieve the same with an iterative program that uses only two local variables:

```
unsigned fib_iterative(unsigned n) {
  unsigned f1 = 1;
  unsigned f2 = 0;
  unsigned f  = n;
  while (n-- > 1) {
    f = f1 + f2;
    f2 = f1;
    f1 = f;
  }
  return f;
}
```

Open in Browser

5.4.2 Computing Binomial Coefficients

The binomial coefficient of two natural numbers $k \leq n$ is defined as:

$$\binom{n}{k} := \frac{n!}{k!(n-k)!} \qquad \text{with } n! := n \cdot (n-1) \cdots 2 \text{ and } 0! := 1 \tag{5.1}$$

To compute the binomial coefficient by the formula above, we will need

$$(n-2) + (k-2) + (n-k-2) + 2 = 2n - 4$$

multiplications. This can be optimized a little by the following observation: The terms $k!$ and $(n-k)!$ are intermediate results when computing $n!$. Hence, we can memorize their value and re-use it in the computation of $n!$ like so:

$$\binom{n}{k} = \frac{n(n-1)\cdots(n-m+1)}{m!} \qquad \text{with } m := \min(k, n-k) \leq n/2$$

Then, we only need $(n-(n-m+1)) + (m-2) + 1 = 2m - 2$ multiplications. This means $n-2$ multiplications in the worst case because $m < n/2$.

However, the nominator and denominator can get quite large when computing the binomial coefficient this way. If we assume that we compute with unsigned ints of 32-bit length, the largest n with $n! < 2^{32}$ is $n = 12$. We can empirically determine that the largest binomial coefficient that we can *compute* with the above equation without overflow is $\binom{17}{8} = 24310$. However, the largest binomial coefficient we can *represent* with 32-bit is way larger.

One way of making the computation of larger binomial coefficients feasible and efficient is to use the following recurrence equation

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \qquad \binom{n}{0} = \binom{n}{n} = 1 \tag{5.2}$$

that directly leads to what is known as Pascal's triangle:

	$k = 0$	1	2	3	4	5	6	7	$\cdots$
$n = 0$	1								
1	1	1							
2	1	2	1						
3	1	3	3	1					
4	1	4	6	4	1				
5	1	5	10	10	5	1			
6	1	6	15	20	15	6	1		
7	1	7	21	35	35	21	7	1	
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	

These recursive equations directly yield a recursive program:

```
unsigned int binomial(unsigned int n, unsigned int k) {
    if (k > n)
        return 0;
    if (k == 0 || k == n)
        return 1;
    return binomial(n - 1, k - 1) + binomial(n - 1, k);
}
```

Open in Browser

However, as in the previous section, some of the partial results are re-computed multiple times, because

$$\binom{n}{k+1} = \binom{n-1}{k} + \binom{n-1}{k+1}$$

and the first summand is also used to to compute $\binom{n}{k}$. Without memoization, this program performs exponentially many calls in n. Memoizing the partial results then yields a more efficient algorithm that we develop in the following.

From (5.1) follows that

$$\binom{n}{k} = \binom{n}{n-k}$$

which is equivalent to saying that Pascal's triangle is symmetrical. This means that we never need to memoize a binomial coefficient larger than $\lfloor n/2 \rfloor$. From (5.2) follows that we don't need to compute binomial coefficients larger than k in order to compute $\binom{n}{k}$. So, to compute $\binom{n}{k}$, we need a two-dimensional table with k columns and n rows. For example, this is the table for computing $\binom{7}{3}$:

	$k = 0$	1	2	3	4	5	6	7	$\cdots$
$n = 0$	1	0	0	0					
1	1	1	0	0					
2	1	2	1	0					
3	1	3	3	1					
4	1	4	6	4	.				
5	1	5	10	10	.	.			
6	1	6	15	20	.	.	.		
7	1	7	21	35	.	.	.	.	

The following code implements the memoization technique to compute binomial coefficients:

```
unsigned int binomial_table(unsigned int N, unsigned int K)
    {
    if (K > N)
        return 0;
    if (K > N - K)
        K = N - K;
    if (K == 0)
        return 1;

    unsigned table[N + 1][K + 1];
    for (unsigned n = 0; n <= N; n++) {
        table[n][0] = 1;
        unsigned const end = n < K ? n : K;
        for (unsigned k = 1; k <= end; k++)
            table[n][k] = table[n - 1][k] + table[n - 1][k
                - 1];
        for (unsigned i = end + 1; i <= K; i++)
            table[n][i] = 0;
    }
    return table[N][K];
}
```

Open in Browser

Note that we could even further optimize this code because in order to compute the k-th row of the table, we only need the previous row. So we actually don't need to memoize all rows but just two and "double-buffer" them.

5.4.3 Computing the Minimum Edit Distance

Let us turn to our first real dynamic programming algorithm. Here, we are not only considering overlapping sub-problems but also *optimal substructure*.

Given two words a and b, the minimum edit distance (sometimes also called the Levenshtein distance) between a and b is the minimum number of edit operations (delete/replace/add a character) to transform a to b. Solving edit distance problems is relevant in fuzzy string search settings, such as spell checking. Also, problems like computing genome sequence alignments are related very closely to edit distance problems.

Let's consider an example:

$$a = \texttt{tore} \qquad b = \texttt{tag}$$

Trivially, we can transform `tore` into `tag` by four delete and three add operations: First delete all letters and then add the three letters `t`, `a`, `g`. This results in seven edit operations. So each sequence of edit operations that takes us from `tore` to `tag` has a certain *cost*, i.e. the number of additions, replacements, and deletions.

However, we are interested in the shortest/minimum sequence of edits which in this example has three edit operations: Keep `t` (no edit operations), replace `o` by `a`, replace `r` by `g` and delete `e`.

Let us discuss the edit distance problem more formally.

Definition 5.4.3 Minimum Edit Distance. The minimum edit distance of two words a and b *up to* positions i and j is defined as follows:

$$\begin{array}{lcl}
edist_{a,b} & : & \{0, \ldots, |a|\} \times \{0, \ldots, |b|\} \longrightarrow \mathbb{N} \\
edist_{a,b}(0, j) & \mapsto & j \\
edist_{a,b}(i, 0) & \mapsto & i \\
edist_{a,b}(i, j) & \mapsto & \min \begin{cases} edist_{a,b}(i-1, j) + 1 & \text{delete} \\ edist_{a,b}(i, j-1) + 1 & \text{add} \\ edist_{a,b}(i-1, j-1) & \text{if } a_i = b_j \qquad \text{keep} \\ edist_{a,b}(i-1, j-1) + 1 & \text{if } a_i \neq b_j \qquad \text{replace} \end{cases}
\end{array}$$

◊

From the recursive definition of *edist* we can see that some partial results are used multiple times. Hence, we will also use to memoization to compute the edit distance of two words efficiently. Let us apply Definition 5.4.3 to the example above.

Let us first consider the top row and left column. The entries of the first row correspond to the costs to transform the empty word to the words ε, `t`, `ta`, `tag` by adding the respective characters.

	ε	t	a	g
ε	0	1	2	3
t				
o				
r				
e				

Accordingly, the entries in the first column corresponds to the costs to get to the empty word from ε, `t`, `to`, `tor`, `tore` by deleting the respective letters.

	ε	t	a	g
ε	0	1	2	3
t	1			
o	2			
r	3			
e	4			

The remaining table entries are filled with the minimum expression of Definition 5.4.3. For example, for the entry at (1, 1), the third case of the case distinction contributes the minimum value:

	ε	t	a	g
ε	0	1	2	3
t	1	2		
o	2			
r	3			
e	4			

	ε	t	a	g
ε	0	1	2	3
t	1	2		
o	2			
r	3			
e	4			

	ε	t	a	g
ε	0	1	2	3
t	1	0		
o	2			
r	3			
e	4			

The rest of the matrix is filled accordingly:

	ε	t	a	g
ε		1	2	3
t	1	0	1	2
o	2	1	1	2
r	3	2	2	2
e	4	3	3	3

According to the definition in Definition 5.4.3, the entry in row i and column j equals to the minimum edit distance of the words $a[1:i]$ and $b[1:j]$. The elements of the first row and column are always the same for each pair of words. All other elements can be computed from the elements above, left, and above left.

Computing the minimum edit distance is an *optimization problem* because we are looking not only for the cost of some sequence of edits but for the *minimum* cost. The property of this problem that makes it amenable to be solved with dynamic programming is that it possesses *optimal substructure*: The *optimal* solution $edist(i, j)$ can be computed from the optimal solutions $edist(i, j{-}1)$, $edist(i{-}1, j)$, $edist(i{-}1, j{-}1)$ of the respective sub-problems.

Chapter 6

Formal Semantics

In this section we give a formal account of a very small subset of C, called C0. We will use mathematics to give precise and unambiguous definitions of what a C0 program is and what it means to execute it. This allows us to precisely define the outcome and effects of a program execution. Essentially, we will develop a framework that allows us to execute (interpret) C0 programs in mathematics. Such a formal description is much more precise than any textual rendition. This formal account will shed more light on notions like pointers, addresses, L- and R-evaluation that we have introduced in the previous section in an informal way. To keep the complexity low, we will only consider a very small subset of C and omit a large part of the "syntactic sugar".

We start with a most basic subset called C0 that only contains a single data type (int), expressions, and a few statements; no types, no functions, no pointers, only global variables.

We first extend C0 to C0p by adding pointers. Thereby, we will formally define the semantics of the pointer operators & and * and clarify the pointer mechanics we have covered informally in the previous section.

Then we will add blocks and their nesting in C0pb. This sheds light on how new containers are allocated, how "dangling pointers" come to be, and how local variables can be hidden by other local variables.

Finally, we will introduce a small type system and see how static typing can be used to identify faulty programs *statically*, i.e. before they crash.

6.1 The Syntax of Programming Languages

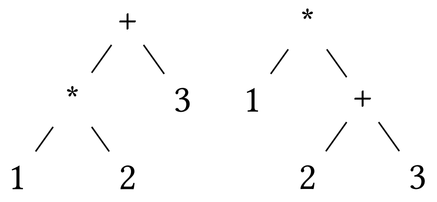
First of all, we need to define what a C0 program actually is. When formalizing the syntax of a programming language, one distinguishes between two different kinds of syntaxes: The **abstract syntax** defines the *structure* of the language's programs. By structure we mean the way how one syntactical construct can be composed from other components (i.e. a while loop consists of a body and an expression). In the abstract syntax, each program corresponds to a tree, the so-called **abstract syntax tree (AST)**. The nodes of the tree correspond to syntactical constructs and the edges denote their composition.

The **concrete syntax** defines how a program in a language can be encoded as a sequence of symbols (characters, digits, etc.). The concrete syntax is relevant for the programmers because they have to type in their programs somehow. When reasoning about programs formally, we focus on the abstract syntax because it captures the *structure* of the program and is not "contaminated" with syntactical artifacts like parentheses and other separators which are only meaningful in a linear (textual) representation of the program.

Remark 6.1.1 Syntax Analysis. The process of obtaining the abstract syntax tree from a textual representation of the program is called **syntax analysis**. Syntax analysis is a core component of each **interpreter** or **compiler** and is usually performed in two separate phases: The first phase is called **lexing** and combines subsequent characters into a stream of **tokens**. Each token stands for a lexical category such as “integer number”, “identifier”, “plus symbol”, and so on. Lexing accounts for the fact that the concrete characters such as the names of variables are not important for syntax analysis because they do not influence the structure of the program. Of course, lexing can already identify some ill-formed programs. For example, it is illegal to put the character sequence `123abc` outside of a comment in a C program. Such errors are identified during lexing.

The second phase is called **parsing**. Parsing constructs the AST from the input token sequence or reports an error if the input token sequence does not lead to a program that adheres to the rules of the concrete syntax.

Remark 6.1.2 Parsing. At this point you may wonder if every program text that is syntactically correct with respect to the *concrete syntax* actually has a unique abstract syntax tree that it corresponds to. The answer is that this depends on the definition of the concrete syntax. For example, if the concrete syntax is defined by the grammar $e \longrightarrow e + e \mid e * e \mid c$, then `1 * 2 + 3` actually has two ASTs:



In this case we say that the concrete syntax is **ambiguous**. To make the concrete syntax in this case unambiguous, we need to add parentheses and encode operator precedence in the grammar.

Checkpoint 6.1.3 Make the grammar of Remark 6.1.2 unambiguous.

The abstract syntax of a language is given by a **syntax definition** using a table like the following:

Category		Abstract Syntax	Concrete Syntax	Description
$Expr \ni e$	$::=$	$\texttt{Add}[e_1, e_2]$	$e_1 + e_2$	Addition
	$\mid$	$\texttt{Mul}[e_1, e_2]$	$e_1 * e_2$	Multiplication
	$\mid$	$\texttt{Const}_c$	c	Constant
	$\mid$	$\texttt{Var}_x$	x	Variable

The column “Category” defines a specific syntactic category (expression, statement, and so on). For each category there may be multiple rows. Each row defines one syntactic construct of the respective category. The column “Abstract Syntax” defines the *structure* for that specific construct: A piece of abstract syntax consists of

- a label (such as `Add`) that denotes the construct. It is also used as the label of a node in an abstract syntax tree that represents the instance of such a construct in a concrete program.
- a list of children given in brackets. These are given by variable names of specific categories, for example $\texttt{Add}[e_1, e_2]$ means that an `Add` construct consists of two sub-expressions.
- possibly some attributes given as subscripts to the label. These attributes constitute additional information for a piece of abstract syntax such as the concrete name of an identifier or the value of a constant. These are not important for

the structure of the program but may be important later on to analyze, execute, or transform the program.

The column "Concrete Syntax" shows how the construct looks like in written program text. In both columns one can use variables which stand for other syntactic constructs with the same name (e.g. e for expressions). Each variable must be defined in the category column to indicate from which category a construct comes. The example above for instance defines a construct `Add` which belongs to the category *Expr* and stands for an addition expression and consists of two other constructs that are from the category *Expr* as well.

Remark 6.1.4 When writing down the abstract syntax of a program on paper, one often resorts to the concrete syntax again instead of drawing an AST because the textual rendition of the AST saves space and looks concise for humans. We will do the same for the rest of this chapter and write down the programs we discuss in concrete syntax although we are always concerned with their abstract syntax. Because we require the concrete syntax to be unambiguous, it is always clear which abstract syntax tree we are referring to.

6.2 The Abstract Syntax of C0

C0 consists of programs that contain statements which in turn can contain expressions. We will first consider programs and statements and then introduce expressions separately later. Note that C0 does not have functions. It is however not too hard to add them and it is a good exercise to work on that after having studied this chapter.

Statements can form lists which we then call programs. A program can be put into a block which then itself is a statement. This allows us to use lists of statements wherever a statement is required, for example in the consequence and alternative of an if-then-else. In the following, we will be exclusively interested in C0 programs in their abstract syntax. We will nevertheless use some elements' concrete syntax (parentheses, braces, etc.) to make it easier to read them for us humans. Definition 6.2.1 defines the abstract syntax of C0 statements and programs.

Definition 6.2.1 C0 Statement Language.

Category		Abstract Syntax	Concrete Syntax	Description
$Prg \ni p$	::=	`Seq` $[s, p]$	$s\ p$	Sequence
	|	`Term`	ε	Empty Program
	|	`Crash`		Crash
$Stmt \ni s$	::=	`Assign` $[l, e]$	$l = e;$	Assignment
	|	`Block` $[p]$	$\{p\}$	Block
	|	`If` $[e, s_1, s_2]$	`if` $(e)\ s_1$ `else` s_2	If
	|	`While` $[e, s]$	`while` $(e)\ s$	Loop
	|	`Abort`	`abort();`	Abort

◊

Note that the empty program is denoted by the empty string ε. In most of the examples, we do not write down ε explicitly. Only in cases where we want to make explicit that we are talking about the empty program, we use ε. For example, we write the program `x = 2;` ε as `x = 2;`. Some statements, like the if statement, contain expressions, which we introduce in the following:

Definition 6.2.2 C0 Expression Language.

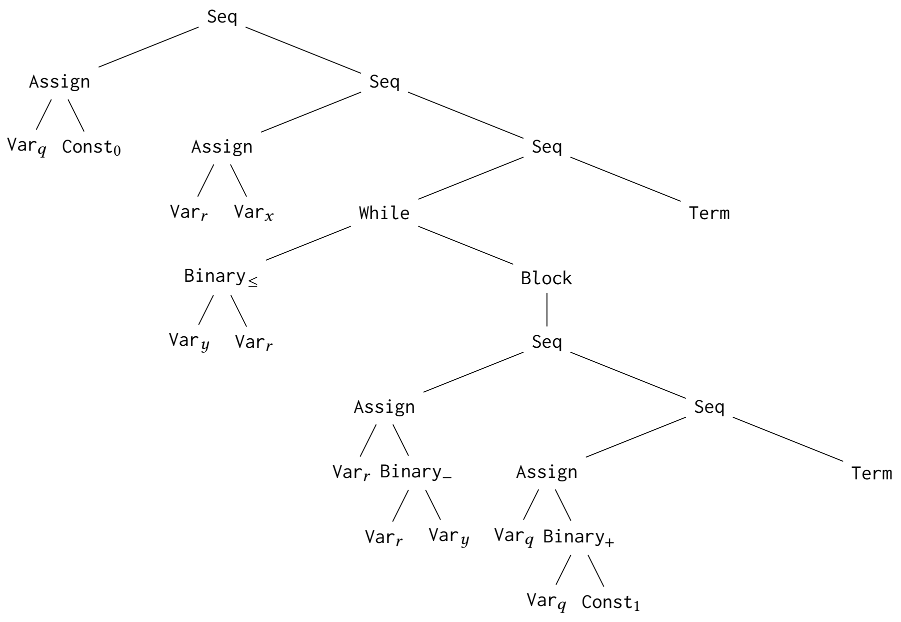
Category		Abstract Syntax	Concrete Syntax	Description
$\mathit{LExpr} \ni l$	::=	$\texttt{Var}_x$	x	Variable
$\mathit{Expr} \ni e$	::=	l		L-Value
	|	$\texttt{Const}_c$	c	Constant
	|	$\texttt{Binary}_o\,[e_1, e_2]$	$e_1\; o\; e_2$	Binary Expr.
$\mathit{Op} \ni o$	::=	$r \mid m$		Operators
$\mathit{AOp} \ni r$	::=		$+ \mid - \mid \cdots$	Arithmetic Op.
$\mathit{COp} \ni m$	::=		$== \mid != \mid < \mid \cdots$	Comparison Op.

◊

Example 6.2.3 The following example shows a small C0 program with its concrete and abstract syntax and the corresponding abstract syntax tree.

```
q = 0;
r = x;
while (y <= r) {
  r = r - y;
  q = q + 1;
}
```

Open in Browser



□

6.3 The Semantics of C0

6.3.1 The State

We have already spoken about the structure of a C program in Section 4.3. Here, we are going to define the notion of a state formally. We assume that there is given a finite set of Variables *Var* and a finite set of addresses *Addr*. We do not assume any further structure on these sets. The set of values is then defined to be $\mathit{Val} := \mathit{Addr} \cup \texttt{int}$.

In Section 4.3, we said that a variable is an identifier that is bound to the address of a container. So, one part of the state is the partial map from variables to addresses:

$$Var \rightharpoonup Addr$$

We call this map **variable assignment**. Note that it is a partial map because not at every time in the program's execution there must be an address bound to a variable: For example, a local variable is only bound to the address of the corresponding container during the execution of its scope.

To keep it simple, we only consider integer variables for now. This means that, in contrast to real C, we do not allow address arithmetic. So in every container there is only a single value. Therefore, we can model containers as a partial map from addresses to values:

$$Addr \rightharpoonup Val \cup \{?\}$$

We call that map **memory assignment**. The value ? is the undefined value which is the content of a container before the container is first assigned a value. Note that the memory assignment is also a a partial map. If some address is not in the domain of a memory assignment, this means that there (currently) does not exist a container with that address.

In summary, the state Σ of a C0 program consists of a pair of partial maps:

$$\Sigma := (Var \rightharpoonup Addr) \times (Addr \rightharpoonup Val \cup \{?\})$$

In our formal discussion we typically use the letter ρ for the variable assignment and the letter μ for the memory assignment. To make the writing a bit more compact we will write the pair (ρ, μ) as $\rho; \mu$. If we do not care about the individual components of a state, we just write it as σ.

Definition 6.3.1 Abbreviated Notation. In many examples that we will discuss in the following, the actual address of a container does not really matter. We will therefore use the short-hand notation $\sigma\, x := (\mu \circ \rho)\, x$ or use the following to specify a state explicitly

$$\{x_1 \mapsto v_1, \dots, x_n \mapsto v_n\}$$

instead of spelling out the variable and memory assignment explicitly

$$\{x_1 \mapsto a_1, \dots, x_n \mapsto a_n\}; \{a_1 \mapsto v_1, \dots, a_n \mapsto v_n\}$$

where the a_i are pairwise different addresses. ◊

6.3.2 Expressions

Intuitively, it is clear how expressions are evaluated. For example, when looking at the expression x + 1, we know from our practical experience that in order to evaluate this expression, first the container that x is bound to is read and then 1 is being added to that value. So in order to evaluate an expression, we need the abstract syntax of the expression itself and a state to lookup the values of the variables, or more precisely get the contents of the containers.

According to our discussion in Subsection 4.8.2, there are two ways to evaluate an expression. The L-evaluation $L[\![\cdot]\!]$ that evaluates *some* expressions to addresses of containers and the R-evaluation $R[\![\cdot]\!]$ that yields actual values. In the basic version of C0 that we have discussed so far, the only L-evaluable expression is the identifier (that denotes a variable) because C0 does not have pointers (yet).

Definition 6.3.2 Expression Evaluation in C0. R-evaluation and L-evaluation are defined by the following two functions

$$R[\![\cdot]\!]\cdot : Expr \to \Sigma \rightharpoonup Val$$

$$L[\![\cdot]\!]\cdot : LExpr \to \Sigma \rightharpoonup Addr$$

$$\begin{array}{lcl} L[\![\mathtt{x}]\!]\rho;\mu & = & \rho\,\mathtt{x} \\ R[\![c]\!]\sigma & = & c \\ R[\![e_1\ o\ e_2]\!]\sigma & = & R[\![e_1]\!]\sigma\ o\ R[\![e_2]\!]\sigma \\ R[\![l\,]\!]\rho;\mu & = & \mu\,(L[\![l\,]\!]\rho;\mu) \end{array}$$

◊

Maybe the last row is the most interesting one. R-evaluating an L-evaluable expression is defined via the L-evaluation, which gives an address. This address is then used to get the container value with the memory assignment μ.

Note that both functions, the one for the L-evaluation and the one for the R-evaluation are *partial*, which means that they are not defined for some combinations of expressions and states. For example, $R[\![e/0]\!]\sigma$ is not defined for any expression e and any state σ. Similarly, $L[\![\mathtt{x}]\!]$ is not defined on any state in which x does not have an image.

6.3.3 Statements

We define the execution of a C0 statement as a sequence of **configurations**. Configurations capture the current state of the execution that consists of a remainder program still to be executed and the current state.

Definition 6.3.3 Configuration. A configuration $c \in Conf := Prg \times \Sigma$ is a pair of a program and a state. ◊

The **semantics** of C0 now determines from which configuration we can step into which. This way, the semantics describes the flow of the program and how the state of the program evolves. Formally, we define the semantics by a *binary relation* $\to$ among configurations. In the literature, this approach is known as **small-step operational semantics**.

There are also big-step semantics which relate inputs to outputs and abstract from the program steps that lead from an input to an output. Therefore, big-step semantics cannot model programs that diverge, i.e. do not terminate because they inherently do not result in a final state. Small-step semantics however capture the detailed execution of each statement in “small” steps. In fact, Definition 6.3.5 can be seen as a big-step semantics of C0 defined on top of a small-step semantics.

Definition 6.3.4 Operational Semantics of C0. We define the small-step semantics $\to \subseteq Conf \times Conf$ of C0 by the following rules:

$$\begin{array}{rll}
\langle l = e; p \mid (\rho,\mu)\rangle \to \langle p \mid (\rho, \mu[a \mapsto v])\rangle & \text{if } R[\![e]\!]\sigma = v \in Val & \text{[Assign]} \\
& \text{and } L[\![l]\!]\sigma = a \in Addr & \\
\langle \mathtt{if}\,(e)\,s_1\,\mathtt{else}\,s_2\ p \mid \sigma\rangle \to \langle s_1\ p \mid \sigma\rangle & \text{if } R[\![e]\!]\sigma \neq 0 & \text{[IfTrue]} \\
\langle \mathtt{if}\,(e)\,s_1\,\mathtt{else}\,s_2\ p \mid \sigma\rangle \to \langle s_2\ p \mid \sigma\rangle & \text{if } R[\![e]\!]\sigma = 0 & \text{[IfFalse]} \\
\langle \mathtt{while}\,(e)\,s\ p \mid \sigma\rangle \to \langle s\ \mathtt{while}\,(e)\,s\ p \mid \sigma\rangle & \text{if } R[\![e]\!]\sigma \neq 0 & \text{[WhileTrue]} \\
\langle \mathtt{while}\,(e)\,s\ p \mid \sigma\rangle \to \langle p \mid \sigma\rangle & \text{if } R[\![e]\!]\sigma = 0 & \text{[WhileFalse]} \\
\langle \{p_1\}\ p_2 \mid \sigma\rangle \to \langle p_1 +\!\!+\, p_2 \mid \sigma\rangle & & \text{[Block]} \\
\langle \mathtt{abort}();\, p \mid \sigma\rangle \to \langle \mathtt{Crash} \mid \sigma\rangle & & \text{[Crash]}
\end{array}$$

The rule [Assign] defines the semantics of the assignment statement. The right-hand side expression is R-evaluated to a value and the left-hand side expression is L-evaluated giving an address. In the resulting state, the content of the container bound to the address is updated to the value. Note that [Assign] only relates two configurations if the R-evaluation or the L-evaluation are actually defined, which may not necessarily be the case.

The next two rules [IfTrue] and [IfFalse] handle the if-then-else statement which either steps to s_1 or s_2 depending on how the conditional expression evaluates. Note again that if the R-evaluation of e is not defined, both rules do not relate any two configurations.

The two while rules [WhileTrue] and [WhileFalse] are similar to the if rules. [WhileTrue] however appends the while loop to the body to "implement" the iterative behavior of while.

The block rule concatenates programs p_1 and p_2 which is denoted by the $+\!\!+$ operator. $+\!\!+$ is defined recursively in the following way:

$$\varepsilon +\!\!+ p := p \qquad (s\ p) +\!\!+ p' := s\ (p +\!\!+ p')$$

◊

Based on this semantics we can now formally define some of the terms we have already used colloquially. To this end, let p to be a C0 program and σ a state.

Definition 6.3.5 Execution Trace. An execution trace (or trace for short) of the program p under σ is a *finite* sequence of configurations

$$\langle p \mid \sigma \rangle = c_1, \quad \ldots, \quad c_n$$

such that $c_i \longrightarrow c_{i+1}$ for all $1 \le i < n$ ◊

Definition 6.3.6 Termination and Divergence. p **terminates** under σ if there is a trace $c_1, \ldots, c_n$ such that there is no configuration c' with $c_n \longrightarrow c'$. We then write $\langle p \mid \sigma \rangle \downarrow c_n$. If $\langle p \mid \sigma \rangle$ does not terminate, we say it **diverges**. ◊

Remark 6.3.7 Kinds of Termination. Based on the definition of program syntax 6.2.1, there are three kinds of how a program p can terminate:

1. Proper termination: $\langle p \mid \sigma \rangle \downarrow \langle \varepsilon \mid \sigma' \rangle$

   The program finished execution properly and leaves behind a final state. There is no remainder program left to be executed.

2. Abort: $\langle p \mid \sigma \rangle \downarrow \langle \texttt{Crash} \mid \sigma' \rangle$

   The program ended in an abort statement which terminates the execution immediately with the "crash" program.

3. Getting stuck: $\langle p \mid \sigma \rangle \downarrow \langle s\ p' \mid \sigma' \rangle$

   In this case the program cannot make another step although there still is a rest of the program to be executed. this may for instance occur if the assignment statement cannot step because the evaluation of one of its constituent expressions is not defined on the particular state. This corresponds to situations where C exhibits undefined behavior.

6.3.4 Digression: Getting Stuck

The reason why a program may get stuck is because $L[\![\cdot]\!]$ or $R[\![\cdot]\!]$ may not be total and the rules [Assign,IfTrue,IfFalse,WhileTrue,WhileFalse] only allow progress if the evaluation of their constituent expressions is defined. To have a fourth outcome of program execution next to termination, divergence, abort is surprising and unpleasant. It is possible to define the semantics of a language in such a way that programs can never get stuck. The simplest way to achieve that is to define some kind of "default" behavior to the cases where our semantics gets stuck. For example, in Java, dividing by 0 causes a specific *exception* to be thrown.

Another way of preventing getting stuck is to determine getting-stuck behavior *statically* by some form of static code analysis, e.g. a type system 6.6. For example, Java mandates that local variables are never read before they are assigned and each Java implementation has to ensure this by some sort of code analysis.

Programming languages in which well-typed programs (so programs that adhere to all the static rules) cannot get stuck are called **type-safe** programming languages. C and C0 are not type-safe.

Let us briefly reflect on how we as programmers experience that a program gets stuck and how we can distinguish it from termination, divergence, and abort. If we execute programs on paper using the rules above 6.3.4, we can easily detect if a program got stuck. In practice however, we typically would use a *compiler* that translates some C/C0 program s to a machine code program $\hat{s}$. The C standard mandates that $\hat{p}$ shows one of the defined behaviors of p if it terminates, aborts, or diverges. However, since C is all about performance, it defines situations where p gets stuck as *undefined behavior*. This means that there are no constraints on $\hat{p}$ as soon as p gets stuck: The machine program may behave in any possible way. Notoriously this means that we cannot tell if p got stuck by observing $\hat{p}$.

Aside: Note that C defines a non-deterministic semantics for some language constructs such that a program started on one state may have multiple different *valid* behaviors.

Example 6.3.8 consider the following C0 program and a piece of MIPS code a hypothetical C0 compiler has generated for our program.

```
if (y == 0)
    x = 42;
while (x != 0)
    if (x == 42)
        abort();
```

Open in Browser

```
        li   $t0 42
        bnez $a1 L2
        move $a0 $t0
        b    L2
L1:     bne  $a0 $t0 L2
        jal  abort
L2:     bnez $a0 L1
```

Open in Browser

Now let us consider some of the executions of that program on different states. The variable x is only written if y is 0. This means that the C0 program gets stuck when trying to evaluate the loop condition on all starting states in which y is unequal to 0. However, depending on the contents of the registers `$a0` (parameter x) and `$a1` (parameter y), the machine code program may terminate, abort or diverge. □

6.3.5 Some Examples

Let us now consider some example programs that we then execute with the C0 semantics that we have just defined.

Example 6.3.9 Swapping. The following program swaps the contents of two variables.

```
t = x; x = y; y = t;
```

Open in Browser

We will now derive the execution trace of this program started on the initial state

$$\rho := \{\mathtt{x} \mapsto \triangle, \mathtt{y} \mapsto \bigcirc, \mathtt{t} \mapsto \Diamond\}$$
$$\mu := \{\triangle \mapsto 1, \bigcirc \mapsto 2, \Diamond \mapsto ?\}$$

The symbols △, ◯, ◊ denote concrete addresses. We use geometric symbols here to emphasize that the address of a container in C/C0 is something different than the address of a memory cell in a computer.

$$\langle \mathtt{t = x; x = y; y = t;} \mid (\rho, \{\triangle \mapsto 1, \bigcirc \mapsto 2, \Diamond \mapsto ?\})\rangle$$

$$
\begin{array}{ll}
\rightarrow\langle \mathtt{x = y;\ y = t;} \mid (\rho, \{\triangle \mapsto 1, \bigcirc \mapsto 2, \lozenge \mapsto 1\})\rangle & \text{[Assign]} \\
\rightarrow\langle \mathtt{y = t;} \mid (\rho, \{\triangle \mapsto 2, \bigcirc \mapsto 2, \lozenge \mapsto 1\})\rangle & \text{[Assign]} \\
\rightarrow\langle \varepsilon \mid (\rho, \{\triangle \mapsto 2, \bigcirc \mapsto 1, \lozenge \mapsto 1\})\rangle & \text{[Assign]}
\end{array}
$$

□

In the last example, we spelled out ρ and μ individually. In the following we will use the short-hand notation from Definition 6.3.1.

Example 6.3.10 Minimum. The following program computes the minimum of x and y and puts it into variable r:

```
if (x < y) r = x; else r = y;
```

Open in Browser

Let us run the program on the state $\{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 2, \mathtt{r} \mapsto ?\}$.

$$
\begin{array}{ll}
\langle \mathtt{if\ (x < y)\ r = x;\ else\ r = y;} \mid \{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 2, \mathtt{r} \mapsto ?\}\rangle & \\
\rightarrow\langle \mathtt{r = x;} \mid \{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 2, \mathtt{r} \mapsto ?\}\rangle & \text{[IfTrue]} \\
\rightarrow\langle \varepsilon \mid \{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 2, \mathtt{r} \mapsto 1\}\rangle & \text{[Assign]}
\end{array}
$$

□

Example 6.3.11 Division. The following program computes the quotient q and the remainder r of the division of a non-negative number x by a positive number y.

```
q = 0;
r = x;
while (y <= r) {
    r = r - y;
    q = q + 1;
}
```

Open in Browser

Let us run the program on the state $\{\mathtt{x} \mapsto 5, \mathtt{y} \mapsto 2, \mathtt{q} \mapsto ?, \mathtt{r} \mapsto ?\}$. To improve the readability, we abbreviate the loop body with S.

$$
\begin{array}{llll}
 & \langle \mathtt{q = 0;\ r = x;\ while\ (y \le r)}\ S & \mid \{\mathtt{q} \mapsto ?, \mathtt{r} \mapsto ?, \mathtt{x} \mapsto 5, \mathtt{y} \mapsto 2\}\rangle & \\
\rightarrow & \langle \mathtt{r = x;\ while\ (y \le r)}\ S & \mid \{\mathtt{q} \mapsto 0, \mathtt{r} \mapsto ?, \mathtt{x} \mapsto 5, \mathtt{y} \mapsto 2\}\rangle & \text{[Assign]} \\
\rightarrow & \langle \mathtt{while\ (y \le r)}\ S & \mid \{\mathtt{q} \mapsto 0, \mathtt{r} \mapsto 5, \mathtt{x} \mapsto 5, \mathtt{y} \mapsto 2\}\rangle & \text{[Assign]} \\
\rightarrow & \langle \mathtt{\{r = r - y;\ q = q + 1;\}\ while\ (y \le r)}\ S & \mid \{\mathtt{q} \mapsto 0, \mathtt{r} \mapsto 5, \mathtt{x} \mapsto 5, \mathtt{y} \mapsto 2\}\rangle & \text{[WhileTrue]} \\
\rightarrow & \langle \mathtt{r = r - y;\ q = q + 1;\ while\ (y \le r)}\ S & \mid \{\mathtt{q} \mapsto 0, \mathtt{r} \mapsto 5, \mathtt{x} \mapsto 5, \mathtt{y} \mapsto 2\}\rangle & \text{[Block]} \\
\rightarrow & \langle \mathtt{q = q + 1;\ while\ (y \le r)}\ S & \mid \{\mathtt{q} \mapsto 0, \mathtt{r} \mapsto 3, \mathtt{x} \mapsto 5, \mathtt{y} \mapsto 2\}\rangle & \text{[Assign]} \\
\rightarrow & \langle \mathtt{while\ (y \le r)}\ S & \mid \{\mathtt{q} \mapsto 1, \mathtt{r} \mapsto 3, \mathtt{x} \mapsto 5, \mathtt{y} \mapsto 2\}\rangle & \text{[Assign]} \\
\rightarrow & \langle \mathtt{\{r = r - y;\ q = q + 1;\}\ while\ (y \le r)}\ S & \mid \{\mathtt{q} \mapsto 1, \mathtt{r} \mapsto 3, \mathtt{x} \mapsto 5, \mathtt{y} \mapsto 2\}\rangle & \text{[WhileTrue]} \\
\rightarrow & \langle \mathtt{r = r - y;\ q = q + 1;\ while\ (y \le r)}\ S & \mid \{\mathtt{q} \mapsto 1, \mathtt{r} \mapsto 3, \mathtt{x} \mapsto 5, \mathtt{y} \mapsto 2\}\rangle & \text{[Block]} \\
\rightarrow & \langle \mathtt{q = q + 1;\ while\ (y \le r)}\ S & \mid \{\mathtt{q} \mapsto 1, \mathtt{r} \mapsto 1, \mathtt{x} \mapsto 5, \mathtt{y} \mapsto 2\}\rangle & \text{[Assign]} \\
\rightarrow & \langle \mathtt{while\ (y \le r)}\ S & \mid \{\mathtt{q} \mapsto 2, \mathtt{r} \mapsto 1, \mathtt{x} \mapsto 5, \mathtt{y} \mapsto 2\}\rangle & \text{[Assign]} \\
\rightarrow & \langle \varepsilon \mid \{\mathtt{q} \mapsto 2, \mathtt{r} \mapsto 1, \mathtt{x} \mapsto 5, \mathtt{y} \mapsto 2\}\rangle & & \text{[WhileFalse]}
\end{array}
$$

□

6.4 Pointers (C0p)

In this section, we extend C0 with pointers. This extension only affects the expression language of C0. The statement language stays the same.

$$\begin{array}{rcllll} \mathit{LExpr} \ni l & ::= & \cdots & \cdots & \text{like defined above} \\ & \mid & \texttt{Indir}\,[e] & *e & \text{Indirection} \\ \mathit{Expr} \ni e & ::= & \cdots & \cdots & \text{like defined above} \\ & \mid & \texttt{Addr}\,[l] & \&l & \text{Address-Of} \end{array}$$

Let us now extend the L- and R-evaluation functions to this extended syntax.

Definition 6.4.1 Pointer Extension to the C0 Expression Evaluation.

$$\begin{aligned} R[\![\&l]\!]\sigma &:= L[\![l]\!]\sigma \\ L[\![*e]\!]\sigma &:= R[\![e]\!]\sigma \qquad\qquad \text{if } R[\![e]\!]\sigma \in \mathit{Addr} \end{aligned}$$

◊

Note that $L[\![*e]\!]\sigma$ may not be defined if $R[\![e]\!]\sigma$ does not yield an address.

This definition also nicely displays the "duality" of both operators. & turns an L-value into an R-value and $*$ does the opposite.

Example 6.4.2 Consider the state:

$$\sigma = \rho; \mu \quad \text{with} \quad \rho := \{\mathsf{x} \mapsto \triangle, \mathsf{y} \mapsto \bigcirc\} \quad \mu := \{\triangle \mapsto \bigcirc, \bigcirc \mapsto 3\}$$

Apparently, x points to the same container as y. y points to a container that contains the value 3. Let us consider the R-evaluation of the expression $*x + y - 1$ using Definition 6.4.1:

$$\begin{aligned} R[\![*\mathsf{x}+\mathsf{y}-1]\!]\sigma &= R[\![*\mathsf{x}+\mathsf{y}]\!]\sigma - R[\![1]\!]\sigma \\ &= R[\![*\mathsf{x}+\mathsf{y}]\!]\sigma - 1 \\ &= R[\![*\mathsf{x}]\!]\sigma + R[\![\mathsf{y}]\!]\sigma - 1 \end{aligned}$$

Let us consider the R-evaluation of the last two remaining terms in detail:

$$\begin{array}{rcl} R[\![\mathsf{y}]\!]\sigma &=& \mu\, L[\![\mathsf{y}]\!]\sigma \\ &=& \mu\,(\rho\,\mathsf{y}) \\ &=& \mu \bigcirc \\ &=& 3 \end{array} \qquad \begin{array}{rcl} R[\![*\mathsf{x}]\!]\sigma &=& \mu\, L[\![*\mathsf{x}]\!]\sigma \\ &=& \mu\, R[\![\mathsf{x}]\!]\sigma \\ &=& \mu\,(\mu\, L[\![\mathsf{x}]\!]\sigma) \\ &=& \mu\,(\mu\,(\rho\,\mathsf{x})) \\ &=& \mu\,(\mu\,\triangle) \\ &=& \mu \bigcirc \\ &=& 3 \end{array}$$

The final result of R-evaluating the above expression is hence 5. □

6.5 Scopes (C0b)

At the moment, there is no way to declare local variables in C0. The program has to start in a state that already contains all variables that the program uses. In this section, we extend C0 to scopes that allow for declaring local variables.

To this end, we modify the block syntax to contain variable declarations. The statement list of a block is preceded by a (possibly empty) list of variable declarations. Each declaration consists of a type and a variable. In this section, the type does not matter yet, but in the next section, we will add a small type system to C0 and make use of the types. The new block rules look like this:

$\mathit{Stmt} \ni s$	$::=$	$\cdots$	$\cdots$	like before
	$\mid$	$\mathtt{Block}_{x_1,\dots,x_m}\,[k_1,\dots,k_m,p]$	$\{k_1\;x_1;\dots k_m\;x_m;p\}$	block with variables
	$\mid$	$\blacksquare$	$-$	dispose local variables

where the k_i are *scalar types*, i.e. either pointers or ints which we formally introduce in Definition 6.6.4. Additionally, we require that the variable names are pairwise different.

To model scoping in the semantics, we need to work a little. As discussed informally in Chapter 4, when entering a block, for each local variable a new container is created. When leaving the block, the containers are de-allocated and the added variables disappear again.

Example 6.5.1 We expect that after running the following program, y has the value 3 and not 2.

```
{
  int x;
  x = 3;
  {
    int x;
    x = 2;
  }
  y = x;
}
```

Open in Browser

□

To realize this behavior, we change our definition of the state to contain a *stack* of variable assignments. When entering the block, a new variable assignment that contains all variables declared in that block will be put on the stack. When leaving the block, the corresponding variable assignment will we removed from the stack. To be able to detect the end of the block in the semantics, we add the statement $\blacksquare$ to the abstract syntax. This syntactical construct is artificial and only used in the semantics. The programmer cannot use it in their programs. Additionally, when entering a block, we also need to provision new containers for the local variables. This is done by the new rule [Scope].

$$\langle \{k_1\;x_1;\dots k_m\;x_m;p_1\}\,p_2 \mid \rho_1,\dots,\rho_k;\mu\rangle \rightarrow \langle p_1 +\!\!+ \blacksquare\,p_2 \mid \rho_1,\dots,\rho_k,\rho_{k+1};\mu'\rangle \quad \text{[Scope]}$$

with

$$\begin{aligned}\rho_{k+1} &= \{x_1\mapsto a_1,\dots,x_m\mapsto a_m\}\\ \mu' &= \mu[a_1\mapsto ?,\dots,a_m\mapsto ?]\\ \emptyset &= \{a_1,\dots,a_m\}\cap \operatorname{dom}\mu\end{aligned}$$

The third condition states that the addresses a_i need to be fresh: They must not exist in the memory assignment.

The leave rule terminates to a state in which the top-most variable assignment is removed from the stack and all addresses added in that block are removed from the memory assignment. These are all addresses in the *image* of the variable assignment that is being removed.

$$\langle \blacksquare\,p \mid \rho_1,\dots,\rho_k,\rho_{k+1};\mu\rangle \rightarrow \langle p \mid \rho_1,\dots,\rho_k;\mu'\rangle \quad \text{[Leave]}$$

with

$$\mu' := \mu|_{\operatorname{img}\rho_1\cup\dots\cup\rho_k}$$

Finally, we need to adapt the functions for L- and R-evaluation to cope with stacks of variable assignments. All but the case for the L-evaluation for variables can be just lifted to stacks of variable assignments. The new L-evaluation for variables using variable assignment stacks just searches for the first occurrence of the variable name in the stack from top to bottom:

$$L[\![x]\!]\rho_1,\dots,\rho_k;\mu = \begin{cases} \rho_k\, x & \text{if } x \in \operatorname{dom}\rho_k \\ L[\![x]\!]\rho_1,\dots,\rho_{k-1};\mu & \text{otherwise} \end{cases}$$

Let us put everything together and work through an example.

Example 6.5.2 We will consider the derivation of the program of the example above on the state

$$\rho = \{\mathtt{y} \mapsto \triangle\} \quad \mu := \{\triangle \mapsto ?\}$$

$$\begin{array}{llll}
 & \langle\texttt{\{int x; x = 3; \{ int x; x = 2; \} y = x;\}} & |\{\mathtt{y}\mapsto\triangle\};\{\triangle\mapsto ?\}\rangle & \\
\to & \langle\texttt{x = 3; \{ int x; x = 2; \} y = x; } \blacksquare & |\{\mathtt{y}\mapsto\triangle\},\{\mathtt{x}\mapsto\bigcirc\};\{\triangle\mapsto ?,\bigcirc\mapsto ?\}\rangle & [\text{Scope}] \\
\to & \langle\texttt{\{ int x; x = 2; \} y = x; } \blacksquare & |\{\mathtt{y}\mapsto\triangle\},\{\mathtt{x}\mapsto\bigcirc\};\{\triangle\mapsto ?,\bigcirc\mapsto 3\}\rangle & [\text{Assign}] \\
\to & \langle\texttt{x = 2; } \blacksquare\ \texttt{ y = x; } \blacksquare & |\{\mathtt{y}\mapsto\triangle\},\{\mathtt{x}\mapsto\bigcirc\},\{\mathtt{x}\mapsto\lozenge\}; & [\text{Scope}] \\
 & & \ \{\triangle\mapsto ?,\bigcirc\mapsto 3,\lozenge\mapsto ?\}\rangle & \\
\to & \langle\blacksquare\ \texttt{y = x; } \blacksquare & |\{\mathtt{y}\mapsto\triangle\},\{\mathtt{x}\mapsto\bigcirc\},\{\mathtt{x}\mapsto\lozenge\}; & [\text{Assign}] \\
 & & \ \{\triangle\mapsto ?,\bigcirc\mapsto 3,\lozenge\mapsto 2\}\rangle & \\
\to & \langle\texttt{y = x; } \blacksquare & |\{\mathtt{y}\mapsto\triangle\},\{\mathtt{x}\mapsto\bigcirc\};\{\triangle\mapsto ?,\bigcirc\mapsto 3\}\rangle & [\text{Leave}] \\
\to & \langle\blacksquare & |\{\mathtt{y}\mapsto\triangle\},\{\mathtt{x}\mapsto\bigcirc\};\{\triangle\mapsto 3,\bigcirc\mapsto 3\}\rangle & [\text{Assign}] \\
\to & \langle\varepsilon \mid \{\mathtt{y}\mapsto\triangle\};\{\triangle\mapsto 3\}\rangle & & [\text{Leave}]
\end{array}$$

□

6.6 A Simple Type System (C0t)

The set of C0 programs that are defined by the abstract syntax in Section 6.2 contains programs that get stuck for some input. For example, the following program is a syntactically-correct C0 program. (Which means that the sequence of characters in the program text complies with the concrete syntax of C0 and therefore we can attribute a unique abstract syntax tree with this text.) However, the execution of the program, as defined by the C0 semantics directly leads to a configuration in which the program gets stuck.

```
{
  int x;
  x = 666;
  *x = 42;
}
```

Open in Browser

The reason is that the contents of x's container is not an address but an integer. For some programs, such as the program above, we can detect and prevent their getting stuck *statically*. We do this by means of a **static semantics**. This semantics gives "meaning" to the abstract syntax of a C0 program, hence the name semantics. This meaning is however not dependent on some input but *independent* of any input, hence it is static. Types of variables and expressions are one example for such a static property.

From the information that variable x has type int, we want to deduce that for *every* program execution x contains an int and never, for example, an address. Hence, ∗x will always get stuck if x has type int. In the following, we will formulate type rules that characterize programs for which such errors do not occur. We will call these programs then **well-typed**.

Since C0 is not a **statically type-safe** language, there will be programs that are well-typed but nevertheless get stuck, for example:

```
int x;
x = x + 1
```

Open in Browser

Listing 6.6.1 x is read from but not written to before.

```
int x;
int *p;
{ int y; p = &y; }
x = *p;
```

Open in Browser

Listing 6.6.2 After the inner block has been left, the pointer p is dangling (i.e. contains an invalid address) because the container of y has been freed.

```
int x = 666;
x = x / 0;
```

Open in Browser

Listing 6.6.3 Division by zero also is undefined behavior in C.

This means that in C0 and C there are getting-stuck situations that are not ruled out by the static semantics. Languages where well-typed programs never get stuck are called **statically type-safe**[12]

The static semantics of C0 is defined by relations that relate the abstract syntax of C0 with a **type environment**. The type environment maps, based on variable declarations, identifiers to types. These relations are defined inductively for each language element of C0 which means that we can elaborate the typing relation for an expression based on the typing relations of its sub-expressions.

Before defining the typing relations for statements and expressions, we need to add types to C0. For the sake of simplicity, we only define the concrete syntax here, since we mostly use concrete syntax for better readability in our formal development.

Definition 6.6.4

Category		Concrete Syntax	Description
$ITy \ni i$	::=	`char` | `int`	integer type
$PTy \ni p$	::=	$t*$	pointer type
$STy \ni k$	::=	$p \mid i$	scalar type
$Ty \ni t$	::=	k | `void`	type

◊

C supports implicit type conversion which means that at some places values of some type can be automatically (without further annotations by the programmer) converted to a value of a different type. For example, a `void`$*$ can be implicitly converted into a `int`$*$ We model this here by the relation $\leftrightarrow$.

Definition 6.6.5 Implicit Type Conversion.

$$\frac{}{i_1 \leftrightarrow i_2}$$

$$\frac{}{t* \leftrightarrow t*}$$

$$\frac{}{t* \leftrightarrow \texttt{void}*}$$

[12]In statically type-safe languages (like C#, Java, OCaml, etc.), the semantics ensures that in exceptional situations that cannot be covered statically because they may depend on the input of a program, a certain well-defined behavior is triggered, such as throwing an exception.

$$\frac{}{\texttt{void}{*} \leftrightarrow t{*}}$$

◊

6.6.1 Expressions

The static semantics of the C0 expression language is defined by a relation

$$\mathit{ExprS} \subseteq (\mathit{Var} \rightharpoonup \mathit{Ty}) \times \mathit{Expr} \times \mathit{Ty}$$

which we will inductively define over the syntax of C0. It is standard to use the following notation to indicate that a triple of type environment, expression, and type is in *ExprS*:

$$\Gamma \vdash e : t \quad :\Longleftrightarrow \quad (\Gamma, e, t) \in \mathit{ExprS}$$

So, $\Gamma \vdash e : t$ says that expression e has type t under type environment Γ.

Definition 6.6.6 Static Semantics of the C0 Expression Language.

$$[\text{TVar}]\ \frac{\Gamma\, x = k}{\Gamma \vdash x : k}$$

$$[\text{TConst}]\ \frac{-N \le c < N \qquad N = 2^{31}}{\Gamma \vdash c : \texttt{int}}$$

$$[\text{TArith}]\ \frac{\Gamma \vdash e_1 : i_1 \qquad \Gamma \vdash e_2 : i_2}{\Gamma \vdash e_1\ r\ e_2 : \texttt{int}}$$

$$[\text{TCmp}]\ \frac{\Gamma \vdash e_1 : k_1 \qquad \Gamma \vdash e_2 : k_2 \qquad k_1 \leftrightarrow k_2}{\Gamma \vdash e_1\ m\ e_2 : \texttt{int}}$$

$$[\text{TPtrArith}]\ \frac{\Gamma \vdash e_1 : k{*} \qquad \Gamma \vdash e_2 : i}{\Gamma \vdash e_1 + e_2 : k{*}}$$

$$[\text{TPtrDiff}]\ \frac{\Gamma \vdash e_1 : k{*} \qquad \Gamma \vdash e_2 : k{*}}{\Gamma \vdash e_1 - e_2 : \texttt{int}}$$

$$[\text{TPtrCmp}]\ \frac{\Gamma \vdash e : t{*}}{\Gamma \vdash e == 0 : \texttt{int}}$$

$$[\text{TPtrCmpN}]\ \frac{\Gamma \vdash e : t{*}}{\Gamma \vdash e\ != 0 : \texttt{int}}$$

$$[\text{TIndir}]\ \frac{\Gamma \vdash e : k{*}}{\Gamma \vdash {*}e : k}$$

$$[\text{TAddr}]\ \frac{\Gamma \vdash l : t}{\Gamma \vdash \&l : t{*}}$$

◊

The rule [TVar] determines the type of the expression 'occurrence of a variable' by looking up the type of the variable in the type environment. Constants are always int. Binary arithmetic expressions also always have the type int irrespective of the operand types. Arithmetic is only allowed for integer types. Pointer arithmetic is handled by two specific rules: [TPtrArith] to add an offset to a pointer and [TPtrDiff] to subtract two pointers of the same type. The results of comparisons are of integer type (executing them yields either 0 or 1). Pointers can also be compared. However, the operands of a comparison have to be implicitly convertible to each other: We can compare a `void`$*$ with a `int`$*$ and a `char` with a `int` but not a `char`$*$ with a `int`$*$. As a special case, pointers can be compared against 0 ([TPtrCmp] and [TPtrCmpN]). Additionally, [TPtrArith], [TPtrCmp], [TCmp], and [TPtrCmpN] also need variants to handle commutativity which we omit here for the sake of brevity.

Note however that comparing two pointers or subtracting two pointers that do not point to the same object is undefined behavior and will cause the program to get stuck. These cases *are not* caught by our type system.

6.6.2 Statements and Programs

Similar to expressions we also use a relation to indicate if a statement and a program is well-typed. Now, statements and programs do not have a type themselves that we can associate with them. The well-typedness relation for statements and programs therefore merely captures if the expressions that are contained in the statements are well-typed. Therefore, the well-typedness relation is only binary and not ternary as the one for expressions:

$$\mathit{StmtS} \subseteq (\mathit{Var} \rightharpoonup \mathit{Ty}) \times \mathit{Stmt} \quad \mathit{PrgS} \subseteq (\mathit{Var} \rightharpoonup \mathit{Ty}) \times \mathit{Prg}$$

and again for better readability and to comply to the standard convention, we use the shorthand notation (similarly for programs):

$$\Gamma \vdash s \quad :\Longleftrightarrow \quad (\Gamma, s) \in \mathit{StmtS}$$

Definition 6.6.7 Static Semantics of C0 Statements and Programs.

$$[\text{TSeq}]\ \frac{\Gamma \vdash s \qquad \Gamma \vdash p}{\Gamma \vdash s\ p}$$

$$[\text{TTerm}]\ \frac{}{\Gamma \vdash \varepsilon}$$

$$[\text{TAssign}]\ \frac{\Gamma \vdash e : k_1 \qquad \Gamma \vdash l : k_2 \qquad k_1 \leftrightarrow k_2}{\Gamma \vdash l = e;}$$

$$[\text{TAbort}]\ \frac{}{\Gamma \vdash \texttt{abort}();}$$

$$[\text{TWhile}]\ \frac{\Gamma \vdash e : k \qquad \Gamma \vdash s}{\Gamma \vdash \texttt{while}\,(e)\,s}$$

$$[\text{TIf}]\ \frac{\Gamma \vdash e : k \qquad \Gamma \vdash s_1 \qquad \Gamma \vdash s_2}{\Gamma \vdash \texttt{if}\,(e)\,s_1\,\texttt{else}\,s_2}$$

$$[\text{TBlock}]\ \frac{\Gamma' = \Gamma[x_1 \mapsto k_1, \ldots, x_m \mapsto k_m] \qquad \Gamma' \vdash p}{\Gamma \vdash \{k_1\,x_1; \ldots k_m\,x_m; p\}}$$

◊

The following rule [TSeqS] summarises two applications of [TSeq] and one application of [TTerm] and makes type derivations on programs more compact.

$$[\text{TSeqS}]\ \frac{\Gamma \vdash s_1 \qquad \Gamma \vdash s_2}{\Gamma \vdash s_1\ s_2\ \varepsilon}$$

The rules [TWhile] and [TIf] make sure that the condition expression they contain have a scalar type (i.e. are not `void`). The rule [TAssign] makes sure that the right-hand side type can be converted to the left-hand side type. This rules out a program like the following:

```
{ int *p; p = 5; }
```

Open in Browser

Most notably however is the rule [TBlock] which administers the type environment: In order for a block to be well-typed, its constituent statements must be well-typed under the type environment that additionally contains the local variables declared in the block. Note that this also models variable hiding: Variables declared in inner blocks hide the declarations of outer blocks.

6.6.3 Examples

Let us consider a couple of examples that put our small type system to work and compute the types of several expressions or check the well-typedness of statements.

Example 6.6.8 Let's consider the derivation of the type of an expression with the type environment $\Gamma := \{\mathsf{x} \mapsto \mathsf{char*}, \mathsf{y} \mapsto \mathsf{int}\}$

$$\text{[TArith]}\ \dfrac{\text{[TArith]}\ \dfrac{\text{[TIndir]}\ \dfrac{\text{[TVar]}\ \dfrac{\Gamma\ \mathsf{x} = \mathsf{char*}}{\Gamma \vdash \mathsf{x} : \mathsf{char*}}}{\Gamma \vdash *\mathsf{x} : \mathsf{char}} \quad \text{[TVar]}\ \dfrac{\Gamma \mathsf{y} = \mathsf{int}}{\Gamma \vdash \mathsf{y} : \mathsf{int}}}{\Gamma \vdash *\mathsf{x} + \mathsf{y} : \mathsf{int}} \quad \text{[TConst]}\ \dfrac{-2^{31} \leq 1 < 2^{31}}{\Gamma \vdash 1 : \mathsf{int}}}{\Gamma \vdash *\mathsf{x} + \mathsf{y} - 1 : \mathsf{int}}$$

□

Example 6.6.9 In this example, we consider the same expression but with a different type environment:

$$\Gamma := \{\mathsf{x} \mapsto \mathsf{char}, \mathsf{y} \mapsto \mathsf{int}\}$$

Here, we can see nicely that we are not able to prove the premise $\Gamma \vdash \mathsf{x} : \mathsf{char*}$ with [Var] because the the type environment does not provide the type `char*` for `x`.

$$\text{[TArith]}\ \dfrac{\dfrac{\text{[TIndir]}\ \dfrac{\text{[TVar]}\ \dfrac{\text{Error}}{\Gamma \vdash \mathsf{x} : \mathsf{char*}}}{\Gamma \vdash *\mathsf{x} : \mathsf{char}} \quad \text{[TVar]}\ \dfrac{\Gamma\ \mathsf{y} = \mathsf{int}}{\Gamma \vdash \mathsf{y} : \mathsf{int}}}{\Gamma \vdash *\mathsf{x} + \mathsf{y} : \mathsf{int}} \quad \text{[TConst]}\ \dfrac{-2^{31} \leq 1 < 2^{31}}{\Gamma \vdash 1 : \mathsf{int}}}{\Gamma \vdash *\mathsf{x} + \mathsf{y} - 1 : \mathsf{int}}$$

□

Example 6.6.10 Finally, let's consider the derivation of the static semantics for a more complex example in which we declare local variables in blocks.

$$\text{[TBlock]}\ \dfrac{\text{[TSeq]}\ \dfrac{\text{[TAssign]}\ \dfrac{\vdots}{\Gamma \vdash \mathsf{x} = 3;} \quad \text{[TSeq]}\ \dfrac{\text{[TBlock]}\ \dfrac{\text{[TSeq]}\ \dfrac{\vdots}{\Gamma[\mathsf{x} \mapsto \mathsf{int*}] \vdash \mathsf{x} = \&\mathsf{y};}}{\Gamma \vdash \{\mathsf{int} * \ \mathsf{x};\ \mathsf{x} = \&\mathsf{y}; \};} \quad \text{[TSeq]}\ \dfrac{\text{[TAssign]}\ \dfrac{\vdots}{\Gamma \vdash \mathsf{y} = \mathsf{x};} \quad \text{[TTerm]}\ \dfrac{}{\Gamma \vdash \varepsilon}}{\Gamma \vdash \mathsf{y} = \mathsf{x};}}{\Gamma \vdash \{\mathsf{int} * \ \mathsf{x};\ \mathsf{x} = \&\mathsf{y}; \}; \mathsf{y} = \mathsf{x};}}{\{\mathsf{x} \mapsto \mathsf{int}, \mathsf{y} \mapsto \mathsf{int}\} =: \Gamma \vdash\ \mathsf{x} = 3; \{\mathsf{int} * \ \mathsf{x};\ \mathsf{x} = \&\mathsf{y}; \}\ \mathsf{y} = \mathsf{x};}}{\emptyset \vdash \{\mathsf{int}\,\mathsf{x};\ \mathsf{int}\,\mathsf{y};\ \mathsf{x} = 3; \{\mathsf{int} * \ \mathsf{x};\ \mathsf{x} = \&\mathsf{y}; \}\ \mathsf{y} = \mathsf{x}; \}}$$

□

6.7 Summary and Further Reading

In this section, we discussed a formal semantics of a very small subset of C called C0. We used a *small-step operational semantics* for formalizing C0's statements. There are other approaches of formalizing semantics as well but small step semantics are very close to the intuitive understanding of program execution. We have particularly paid attention to modelling all different program behaviors (terminating, aborting, diverging, getting stuck). We will come back to this when discussing formal verification later on.

C0 leaves out, by choice, many aspects of C which make formalizing C more involved, such as a realistic memory model, side effects in expressions, and so on. All these details are covered by more comprehensive formalizations of C in the dissertations of Norrish [8] and Krebbers [9] and in the verified C compiler CompCert [10].

Very good introductions to formal semantics of programming languages are the textbooks by Nielson and Nielson [5] and Winskel [7]. More advanced topics are covered in the book by Harper [6] and, with more focus on type systems, by Pierce [11].

Chapter 7

Testing and Verification

In this section, we investigate measures to make software more reliable. We first define formal notions of program correctness based on specifications. Then we discuss what failing programs are and how we can detect failures. Afterwards we investigate different techniques to test programs. Finally, we discuss how our formal notions of correctness can be leveraged to design an automated program verification tool that is based on off-the-shelf solvers for logical formulas.

7.1 Functional Correctness

What does it mean if we say that a program is correct? First, there may be different criteria of correctness: The program delivers its results in a certain time budget, which is very important in real-time systems[13] or does only consume a certain amount of memory or does not exhibit certain security problems (e.g. side channels). Most important however is **functional correctness**. By functional correctness, we mean that the program computes what it is supposed to compute. But how to we declare what is the right thing to compute? To this end, we need a **specification**.

Definition 7.1.1 Specification, semantically. A specification $S \subseteq \Sigma \times \Sigma$ is a set of pairs of states. ◊

The idea of this definition is that a specification relates starting states with terminating states of program executions. Each pair in the set of a semantical specification basically says: If you start executing the program in the first state of the pair, program execution shall end in the second state of the pair.

For example, the following specification specifies that variable y has the value $\mathtt{x} + 1$ in the final state.

$$S_{\text{add}} = \{(\sigma, \sigma') \mid \sigma'\,\mathtt{y} = \sigma\,\mathtt{x} + 1\}$$

Let us consider the specification $S_{\max}$ of the computation of the maximum of two variables x and y. It is given by the following set of state pairs:

$$\begin{aligned} S_{\max} = \{ & (\{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 1\}, \{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 1, \mathtt{m} \mapsto 1\}), \\ & (\{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 2\}, \{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 2, \mathtt{m} \mapsto 2\}), \\ & \vdots \\ & (\{\mathtt{x} \mapsto 42, \mathtt{y} \mapsto 37\}, \{\mathtt{x} \mapsto 42, \mathtt{y} \mapsto 37, \mathtt{m} \mapsto 42\}), \end{aligned}$$

[13] think of the software that causes your airbag to unfold.

$$\vdots$$

or more compact:

$$S_{\max} = \{(\sigma, \sigma') \mid \sigma' \, \mathtt{m} \geq \sigma \, \mathtt{x} \wedge \sigma' \, \mathtt{m} \geq \sigma \, \mathtt{y} \wedge (\sigma' \, \mathtt{m} = \sigma \, \mathtt{x} \vee \sigma' \, \mathtt{m} = \sigma \, \mathtt{y})\} \tag{7.1}$$

Ultimately, we want to document the specification of a program *in* the program text itself. So, writing down specifications on the semantical level (i.e. by directly referring to states) is not appropriate because states are not an element of the syntax of a program but its semantics. Therefore, it is common practice to use **assertions** to express constraints on states at a particular program point and use them to express specifications.

Definition 7.1.2 Assertion. An assertion a is a logical formula over the expression language of the programming language. In the following, we will discuss assertions in the context of C0 and define assertions here using the following syntax definition (for the sake of brevity, we only give concrete syntax):

$$\mathit{Assert} \ni A ::= e \mid \mathit{true} \mid \mathit{false} \mid \neg A \mid A_1 \wedge A_2 \mid A_1 \vee A_2 \mid A_1 \Rightarrow A_2 \mid A_1 \Leftrightarrow A_2 \mid \forall x.A \mid \exists x.A$$

where operators more to the left bind stronger. Note that $e \in \mathit{Expr}$ is a C0 expression. ◊

Definition 7.1.2 defines assertions as logical formulas of first-order logic over the C0 expression language. After defining the syntactic structure of an assertion, let us define which set of states is actually described by an assertion. To this end, we define the notion of a **model** of an assertion.

Definition 7.1.3 Model. A state σ is a *model* of an assertion A if it *satisfies* A. Satisfaction is defined by the relation $\vDash$ which is defined inductively on the assertion language:

$$\begin{array}{ll} \sigma \vDash e & \text{iff } R[\![e]\!]\sigma \neq 0 \\ \sigma \vDash \neg A & \text{iff } \sigma \nvDash A \\ \sigma \vDash A_1 \wedge A_2 & \text{iff } \sigma \vDash A_1 \text{ and } \sigma \vDash A_2 \\ \sigma \vDash \forall x.A & \text{iff for every } i \in \mathtt{int} \text{ there is } \sigma[i/x] \vDash A \end{array}$$

All the other operators in *Assert* are just syntactical sugar

$$\begin{array}{rlrl} \mathit{false} &= x \wedge \neg x & \mathit{true} &= \neg \mathit{false} \\ A_1 \vee A_2 &= \neg(\neg A_1 \wedge \neg A_2) & A_1 \Rightarrow A_2 &= \neg A_1 \vee A_2 \\ A_1 \Leftrightarrow A_2 &= (A_1 \Rightarrow A_2) \wedge (A_2 \Rightarrow A_1) & \exists x.A &= \neg \forall x.\neg A \end{array}$$

and $\vDash$ is defined on them accordingly. ◊

Definition 7.1.4 Satisfiability and Validity. We say an assertion A is

- **satisfiable** if it has some model.
- **unsatisfiable** if it has no model.
- **valid** if every state is a model of A which is equivalent to saying that $\neg A$ is unsatisfiable.
- **not valid** if there is a state that is not a model of A which is equivalent to $\neg A$ is satisfiable.

◊

We will now use the assertion language we defined and express specifications *syntactically* by means of a **precondition** and a **postcondition**.

Definition 7.1.5 Specification, syntactically. A pair of assertions (P, Q) is called a **specification**. In such a specification, P is called **precondition** and Q is called **postcondition**. ◊

For example, the assertions

$$P = \mathit{true} \qquad Q = \mathtt{m} \geq \mathtt{x} \land \mathtt{m} \geq \mathtt{y} \land (\mathtt{m} = \mathtt{x} \lor \mathtt{m} = \mathtt{y}) \tag{7.2}$$

constitute a specification for a program that computes the maximum of x and y.

When looking at the semantical (7.1) and syntactical (7.2) specifications for the maximum program, one observes that the semantical specification draws a connection from the starting state to the final state of an execution, e.g. by stating that $\sigma' \, \mathtt{m} \geq \sigma \, \mathtt{x}$: both, σ and σ' appear in (7.1). This is not the case for the syntactical specification that merely consists of two assertions.

So, in principle, the syntactical specification is weaker and admits programs like the following:

```
x = 1;
y = 2;
m = y;
```

Open in Browser

This is not a problem if we are just interested in *verifying* a given program with respect to a specification. However, if we are interested in *synthesis*, i.e. creating programs that are correct with respect to a given specification, this plays a role. We then need additional constraints outside of the specification such as demanding that the program we are looking for does not modify its inputs.

One way of achieving this is introducing **ghost variables** that capture the initial values of all variables in the program but are forbidden to be modified by the program itself. We denote the ghost variables here with capital letters.

Using ghost variables, we can refine the specification of programs that compute the maximum of two values such that the result variable m contains the maximum of values of x and y *at the start of the program.*

$$P = \mathtt{X} = \mathtt{x} \land \mathtt{Y} = \mathtt{y} \qquad Q = \mathtt{m} \geq \mathtt{X} \land \mathtt{m} \geq \mathtt{Y} \land (\mathtt{m} = \mathtt{X} \lor \mathtt{m} = \mathtt{Y}) \tag{7.3}$$

Now, we can use our notion of a specification and define what it means for a program to be correct. We do this by leveraging our formal notion of program semantics that we have developed in Chapter 6 and link it to the formal notion of specifications we developed above.

Definition 7.1.6 Total Correctness. A program p is *totally* correct with respect to a specification (P, Q) if for every model σ of P there is a model σ' of Q such that $\langle p \mid \sigma \rangle$ *properly* terminates 6.3.7 in σ'. More formally:

$$(\sigma \vDash P) \Rightarrow (\exists \sigma'. \langle p \mid \sigma \rangle \downarrow \langle \varepsilon \mid \sigma' \rangle \land \sigma' \vDash Q)$$

It is customary to write the statement "Program p is totally correct with respect to the specification (P, Q)" using a so-called **Hoare triple**[14] $[P] \; p \; [Q]$ ◊

Note that if a program gets stuck or aborts on a state that satisfies the precondition, the program is not totally correct.

Reasoning about total correctness implies reasoning about program termination as well: If we want to prove a program totally correct, we have to implicitly also prove

[15]Named after its inventor Tony Hoare. Initially, Hoare used a different notation and considered only partial correctness.

that it terminates on all states that satisfy the precondition. This is sometimes a bit too heavy and therefore one often uses the notion of partial correctness in practice, where reasoning about termination is left out.

Definition 7.1.7 Partial Correctness. A program p is *partially* correct with respect to a specification (P, Q) if for every model σ of P it either properly terminates in a state σ' that satisfies Q or diverges. More formally:

$$(\sigma \vDash P) \Rightarrow (\forall p'\ \sigma'. \langle p \mid \sigma \rangle \downarrow \langle p' \mid \sigma' \rangle \Rightarrow (p' = \varepsilon \wedge \sigma' \vDash Q))$$

We use the notation $\{P\}\ p\ \{Q\}$ to denote partial correctness. ◊

Note that, alike total correctness, if a program gets stuck or aborts on a state that satisfies the precondition, it is not partial correct.

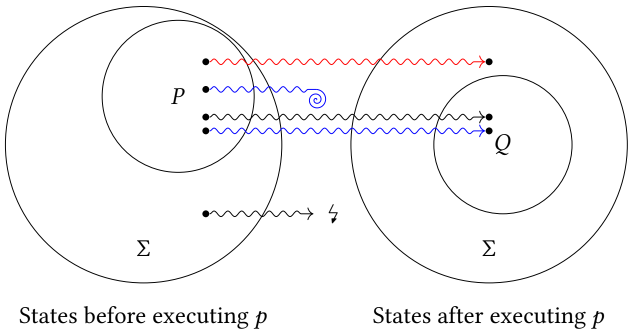


Figure 7.1.8 The set of all states before and after the execution of p. The circle labelled P denotes all states that satisfy the precondition P. Similarly, the circle labelled Q denotes all states that satisfy the postcondition Q. The colors denote different programs and the squiggly lines denote all executions of the program with respect to the particular color.

Consider Figure 7.1.8. The red program is neither partially nor totally correct because it terminates in a state that does not satisfy the postcondition. The blue program is not totally correct because it is not properly terminating in a state for one input. But it is partially correct, because the properly terminating executions end in states that satisfy the postcondition. The black program is totally and partially correct although there is an execution that aborts. This execution however starts in a state that does not fulfill the precondition.

7.2 Failures

Let us consider a program p and a specification (P, Q). If p is not *partially* correct with respect to (P, Q) the program has an **error**. Then, due to Definition 7.1.7, there exists a state $\sigma \vDash P$ such that $\langle p \mid \sigma \rangle$ has one of the following outcomes:

1. $\langle p \mid \sigma \rangle$ gets stuck.
2. $\langle p \mid \sigma \rangle$ aborts.
3. $\langle p \mid \sigma \rangle$ terminates properly in a state σ' but $\sigma' \nvDash Q$.

Definition 7.2.1 Failure. A state $\sigma \vDash P$ is called a **failure** with respect to p and (P, Q) if one of the three cases is true. ◊

Remark 7.2.2 If p gets stuck, the semantics of the machine program that the compiler created for p is undefined. In that case, this machine program can either (a) diverge, (b) abort, or terminate properly in a state that either (c) satisfies or (d) does not satisfy the post condition. This means that, in the case of undefined behavior, we might not be able to observe a failure! This is the case in (a) and (c). So, when testing or debugging programs it is very helpful to avoid getting stuck (i.e. running into undefined behavior). This can be achieved by using sanitizers which instrument the C program to detect undefined behavior during execution and cause the program to abort in such cases.

Remark 7.2.3 If a program is partially but not totally correct, there is an input that satisfies the precondition but causes the program to diverge. That means that if the program is free of failures, we can only conclude that it is *partially* correct.

A failure is a witness for an error. In general however, failures do not inform us about the **cause** of the error, i.e. the part of the program that is erroneous. A failure is therefore a symptom not a diagnosis. Let us consider the following example:

```
unsigned min(unsigned x, unsigned y) {
    if (x < y)
        return y;
    else
        return x;
}

unsigned sum_first_n(unsigned *arr, unsigned sz, unsigned
    n) {
    unsigned sum = 0;
    n = min(n, sz);
    for (unsigned i = 0; i < n; i++)
        sum += arr[i];
    return sum;
}
```

Open in Browser

Assume that the specification is given, as usual in practice, in plain English:

"The function `sum_first_n` shall sum up the first `n`, however at most `sz`, elements of the array `arr`."

Let us consider the following calls to `sum_first_n`:

```
int main() {
     unsigned arr[] = { 1, 2, 3 };
     unsigned r1 = sum_first_n(arr, 3, 2); // (1)
     unsigned r2 = sum_first_n(arr, 3, 3); // (2)
     unsigned r3 = sum_first_n(arr, 3, 4); // (3)
```

Open in Browser

1. The first call is a failure because our specification demands that the result `r1` must be 3, it is however 6.
2. is not a failure.
3. causes the program to get stuck. The last iteration of the loop in `sum_first_n` intends to access the array element `arr + 3` which does not exist and therefore the address `arr + 3` is not valid. We discussed before 7.2.2 that we cannot make any assumptions on the program's behavior in this case.

The **cause** for the failure in (1) and the getting stuck in (3) is an *error* (also called defect or bug) in the function `min`: `min` does not compute the minimum but the maximum.

This examples shows that a failure may tell us little about the cause of the error that failure exposes. In practice, it can be especially hard to localize the error in program code. The first input merely results in a wrong result whereas the third causes undefined behavior which can be hard to observe (see Remark 7.2.2).

Finding errors in programs can be significantly simplified if programs fail as early as possible when they enter an erroneous state. Then, the program won't continue executing with "wrong" values and thereby dislocating the cause from the symptom. A very common way to achieve this is to use assertions in the code to *assert* **invariants**, i.e. conditions on the program state at a particular program point that have to hold for *every* input. These assertions can then be checked at runtime and if they are violated, the program can be aborted immediately.

Definition 7.2.4 assert(). We define the function `assert`(e) as `if` (e) {} `else` `abort`();. ◊

In C, assertions are typically implemented as macros that are expanded to an if-then-else construct as in Definition 7.2.4. When defining the macro `NDEBUG` (typically by given the command line switch `-DNDEBUG`) asserts can be deactivated by defining the macro `assert` to the empty text. This is typically done for production code that has been "sufficiently" tested.

Example 7.2.5 Assertions in the Minimum Program.

```
#include <assert.h>
unsigned min(unsigned x, unsigned y) {
    unsigned m = x < y ? x : y;
    assert(m <= x && m <= y && (m == x || m == y));
    return m;
}
```

Open in Browser

`assert` evaluates the argument expression. If the resulting value is 0, then the execution aborts immediately. Hence, the program aborts very close to the error location. □

In general, it is advisable to document as much invariants as possible using assertions. They come for free in terms of runtime in the production code because they can just be disabled. They can help to expose bugs and can make it easier to fix them. And they are also helpful for documentation purposes because the denote (at least a part) of the invariant that the programmer had in mind for the particular program point. This holds especially for pre- and postconditions. Note however that the C expression language is often not strong enough to formulate powerful invariants because it does not contain quantifiers like the assertion language we have defined in Definition 7.1.2 (they would of course be very hard to check at runtime). Therefore, often only a part of the invariant can be documented in an assertion.

Let us consider the following example where we can use assertions productively to specify the precondition. Which in this case is that the array is non empty:

Example 7.2.6 Consider a function that is supposed find the minimum of all elements of an array. This operation is not well defined on empty arrays. So it should not be possible to call the function with an empty array. We document this by the assertion `assert(n > 0);`. In debug builds, this assertion is checked during runtime and readers of the code will see that the precondition is that we don't pass empty arrays.

```
unsigned min_arr(unsigned* arr, unsigned n) {
    assert(n > 0);
    unsigned res = arr[0];
    for (unsigned i = 1; i < n; i++)
        res = min(res, arr[i]);
    return res;
}
```

Open in Browser

Note that checking the assertion in debug builds has a concrete benefit: If we accessed the first element of an empty array, the behavior of the program would be undefined with all the consequences outlined in Remark 7.2.2. Checking the assertion at runtime however directly aborts the program in this situation and the failure we get from that brings us closer to the error location. □

Remark 7.2.7 Defensive Programming. Programming *defensively* means that one does not only consider the **happy path**, that is the simplest situation in which a piece of code is supposed to work in which no exceptional or erroneous situations occur. Defensive programming means that one considers *all* possible inputs also those that violate the precondition. This essentially means that one carefully makes sure that all preconditions are met.

Often, this is misunderstood to mean that the code should not abort in any case, i.e. that it is best, if the precondition is *true*. Some programmers try to make this happen by "inventing" default values that are returned in the error case. In principle, it is a design choice where errors shall be handled.

Consider the following example code. The function `min_idx` shall return the index of a smallest elements in an array. Of course, the function is not defined on empty arrays, because they don't have a minimum. The programmer could indicate the error situation by returning -1 or something similar:

```c
int min_arr(int* arr, unsigned n) {
    if (n == 0)
        return -1;
    int min = arr[0];
    int idx = 0;
    for (unsigned i = 1; i < n; i++) {
        if (arr[i] < min) {
            min = arr[i];
            idx = i;
        }
    }
    return idx;
}
```

Open in Browser

Here, returning -1 does not really help. That -1 is not the minimum of the array. Instead of checking for the result being -1 *after* calling `min_arr`, the programmer could have checked for the length of the array being greater than 0 equally well. There is nothing gained from making `min_arr` "work" on all inputs. On the contrary, there is the risk of moving a possible error further away from the failure. Here, it would be best to just start `min_arr` with an assertion:

```c
int min_arr(int* arr, unsigned n) {
    assert (n > 0);
    ...
}
```

Open in Browser

7.3 Testing

Testing is a technique for finding bugs in programs. Testing means to construct inputs to the program (or a part of a program such as a module or a single function), the so-called **subject under test**, executes the program on these inputs, observes its behavior and judges if the outcome was correct. If the outcome was not correct we say that the test failed. If a test failed, we discovered a bug in the program that we can repair. So, a failing test is a witness of a bug. Having an appropriate (in extent and quality) **test suite** is an important ingredient for increasing the quality for software. However, using tests, we cannot *prove* the correctness of a program. If all tests of a test suite pass, it does not imply that the program is correct[15]:

> Testing shows the presence, not the absence of bugs.
>
> —Edsger Dijkstra

Definition 7.3.1 Test Case. Given a program s and a specification (P, Q). A state σ that satisfies the precondition is a **test** of s. A test is **passed** if $\langle s \mid \sigma \rangle \Downarrow \langle \varepsilon \mid \sigma' \rangle$ and $\sigma' \vDash Q$. If σ is a failure we say the test **failed**. A set of tests is called a **test suite**. ◊

Example 7.3.2 Maximum. Let us consider the specification of the maximum program from (7.2) and the following C0 program:

```
if (x > y)
  m = x;
else
  m = x;
```

Open in Browser

This program passes the test $\sigma_{\checkmark} = \{\mathtt{x} \mapsto 2, \mathtt{y} \mapsto 1\}$ and fails the test $\sigma_{\times} = \{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 2\}$. □

In a concrete programming language, it is hard to just specify a state. Therefore, in testing, we typically use a program to create the state that constitutes the test. This program is then often also called "test". This test then calls the subject under test and checks if the postcondition is satisfied:

```
void test1() {
  int x = 2;
  int y = 1;
  int m = max(x, y);
  assert (m == x);
}
```

Open in Browser

It is very common to test individual modules/units of a program separately. We then speak of **unit testing**. A unit can be a single function or multiple functions that implement parts of a self-contained part of the code. For example, for a binary search tree (see Subsection 5.2.2), a unit test could be inserting an element into the tree and checking that it has indeed been inserted at the right place. A unit test could also be inserting a couple of elements and the iterating over the tree to see if all inserted elements have been visited.

Unit tests are very helpful to isolate failures in the individual modules of a larger system. Without unit tests, bugs in module A may very easily cause failures when running code of module B. As discussed in Section 7.2, finding the defect is harder if the failure is far away from the cause. Unit tests help because they consider only one module and therefore make finding the bug's cause easier. Furthermore, they help in preventing the introduction of **regressions**: When working with a version control system, running unit tests can be automated and new commits that fail existing unit tests can be automatically prevented from entering the central code repository.

In addition to unit tests one also uses **integration tests** to test the interaction among different units. **System tests** test the entire program end-to-end. The notions of integration and system test do not denote specific test techniques but delineate the extent of a test.

[15] unless in the unrealistic theoretical case where the test case really covers every input/output pair.

In general, there are two different paradigms of testing:

Specification-based Testing (black-box testing)	Tests are written only with respect to a specification. The code of the subject to test does not have an influence on the test itself. The advantage is that black-box testing can, because it is based on a specification, uncover missing code if some cases have not been considered or implemented. On the other hand, because it ignores the structure of the program to test, it may be that some code paths are not covered by a black-box test suite.
Structural Testing ("white-box"/glass-box testing)	uses the structure of the source code to design tests that execute as much code of the subject under test as possible. Based on specific metrics (see Subsection 7.3.3) one tries to cover (i.e. execute) as much of the code of the subject under test as possible. However, even if a program passes a test suite with good coverage, it doesn't not necessarily mean that it is free of significant bugs because white-box testing can only test what is there and can't test code that is missing.

7.3.1 The Oracle Problem

A mechanism that decides if a test passes or fails is called a **test oracle**. In the ideal setting we have a specification that can serve as the test oracle as defined in Definition 7.3.1. In practice it is often the case that we do not have a formal specification (as defined in Definition 7.1.5). Often, the specification is only available in prose or even only available in the brains of the developer. However, some parts of the specification may be documented using assertions in the program as suggested in Remark 7.2.7.

In such cases, a typical specification is $(\mathit{true}, \mathit{true})$. Although it looks very weak, this specification at least requires that the program must neither get stuck, nor abort, nor diverge which is sort of the minimum requirement for a properly functioning program. If some parts of the specification are documented using assertions in the source code, these assertions will make tests fail that trigger them which results in a failure whose cause we can fix.

Note that the precondition *true* is also satisfied by states that do *not* satisfy the actual *implicit* specification that is only available in prose or maybe even only implicitly available. When executing the program on such a state we may observe behavior that violates the postcondition *true*, i.e. getting stuck, abort, failure. However, in that case this does not necessarily indicate a bug because the initial state violated the implicit precondition. Therefore, good software aborts if preconditions are violated.

7.3.2 Black-box Testing

As mentioned before, black-box tests are written with respect to a specification. This means they can be written even before the subject under test has even been created. Let us discuss a typical black-box scenario by means of an example (for more details, refer to [13]).

Example 7.3.3 Given is the specification of a function that is supposed to compute the roots of a quadratic polynomial:

"The function `unsigned roots(double a, double b, double c, double* r);` shall compute the real roots of the polynomial $ax^2 + bx + c$. It shall return the number of real roots. If the polynomial has more than two roots, the value `UINT_MAX` shall be returned. The parameter `r` points to an array that can hold at least two doubles. The function `roots` places the found roots into the array pointed to by `r`."

Based on the mathematics of quadratic functions, we know that there are several different cases that we must cover.

1. The polynomial is equal to the zero polynomial ($a = b = c = 0$). Then, `roots` has to return `UINT_MAX`
2. The polynomial is linear ($a = 0$). Then, it has one ($b \neq 0$) or no ($b = 0$) root.
3. The polynomial is quadratic ($a \neq 0$). Then there is either none, one, or two roots.

□

It is typical for black-box testing to *partition* the input space along such case distinctions and then create suitable unit tests for each partition. For our example, the following test suite is suitable but not really sufficient. You can improve the test suite by developing more tests that cover more partitions.

```
void test_null(void) {
    unsigned res = roots(0, 0, 0, NULL);
    assert(res == UINT_MAX);
}

void test_linear(void) {
    double r[2];
    unsigned res = roots(0, 2, 2, r);
    assert(res == 1 && r[0] == -1);
}

void test_linear_none(void) {
    double r[2];
    unsigned res = roots(0, 0, 2, r);
    assert(res == 0);
}
```

Open in Browser

7.3.3 White-box Testing and Coverage

When doing structure-based testing, the **coverage** plays a crucial role. The coverage of a test designates the part of the subject under test that is being executed by the test. There are different kinds of coverage metrics that we briefly discuss here.

Definition 7.3.4 Statement Coverage. A statement of the program is covered by a test σ, if the statement is executed in the trace of σ. ◊

Remark 7.3.5 In C0, branch and statement coverage is the same: for every branch there is a statement whose execution also implies the execution of the corresponding branch. For languages that allow for ifs without elses, this does no longer hold:

```
r = 0;
if (x < 0)
    r = -x;
/* next statement */
```

Open in Browser

Remark 7.3.6 The set of paths through a program may be enormously large if not infinite (if the program has loops). Therefore, it is very hard to achieve high path coverage in practice.

Example 7.3.7 Let us consider the following, erroneous implementation of the specification of Example 7.3.3.

```
unsigned roots_buggy(double a, double b, double c, double
    *r) {
    if (a != 0) {                       // 1
        double p = b / a;               // 2
        double q = c / a;               // 3
        double d = p * p / 4 - q;       // 4

        if (d > 0) {                    // 5
            r[0] = -p / 2 + sqrt(d);    // 6
            r[1] = -p / 2 - sqrt(d);    // 7
            return 2;                   // 8
        }

        r[0] = -p / 2;                  // 9
        return 1;                       // 10
    }

    if (b != 0) {                       // 11
        r[0] = -b / c;                  // 12
        return 1;                       // 13
    }

    return 0;                           // 14
}
```

Open in Browser

The tests of Example 7.3.3 cover the following statements: 1, 11, 12, 13, 14. The covered paths are:

`test_null`	1,11,14
`test_linear`	1,11,12,13
`test_linear_none`	1,11,14

The first test fails and exposes the bug that the zero-polynomial case is missing. The second test passes but does not find the bug in statement 12 which should be `r[0] = -c / b;`. The issue here is that the input $\{\mathtt{a} \mapsto 0, \mathtt{b} \mapsto 2, \mapsto 2\}$ is not well chosen because it cannot detect the swapping of divisor and dividend. A better test would be:

```
void test_linear(void) {
    double r[2];
    unsigned res = roots(0, 4, 2, r);
    assert(res == 1 && r[0] == -0.5);
}
```

Open in Browser

`test_linear_none` passes because the code handles the case of a constant polynomial unequal to 0 correctly. None of the tests exposes that the quadratic case is not correctly implemented: If the discriminant is less than 0, the function erroneously returns a root with the value of $-p/2$.

Also, this test suite does not have a good statement coverage:

$$\frac{|\{1, 11, 12, 13, 14\}|}{|\{1, \ldots, 14\}|} = \frac{5}{14} \approx 35,7\%$$

□

Remark 7.3.8 Test suites that don't cover all statements are unsatisfactory.

Once again, let us emphasize that high coverage does not detect the situation that some functionality has not been implemented because it just means that a high proportion of the code that *is there* is covered. So, high coverage is necessary for a good test suite but not sufficient.

7.3.4 Fuzzing

Finally, we would like to briefly mention fuzz testing. In fuzzing one creates a program that more or less randomly creates inputs for the subject under test and checks if the output crashes or produces a sensible output. Crucial to fuzzing is the ability to generate inputs that satisfy the precondition. For example, producing a text input that adheres to a specific grammar or file format.

Fuzzing is a black-box technique because the actual implementation has no influence on how the inputs are generated. Depending on how complicated the specification is, it can be very hard to come up with an appropriate test oracle for a fuzz tester. So, very often, fuzzing is used to test for specifications like "the program doesn't abort or get stuck".

Furthermore, a very common thing is to use another (maybe simpler and maybe less efficient) implementation of a program A to fuzz test a more complicated implementation of the same problem B. For example, one could use a simple list-based implementation of a set to fuzz test a more complicated implementation of a set that uses binary trees.

7.4 Verification

In this section, we discuss Hoare logic, a verification technique to *statically* verify that a program is correct with respect to a given specification.

We have already discussed Hoare triples as a notation to say that a program is partially/totally correct. We will now extend this notation to a logic, i.e. a set of rules on the abstract syntax of C0 programs with which we can derive statements (Hoare triples) about them. The overall idea is to define appropriate rules to derive Hoare triples for each syntactical element of C0. These rules can then be used to derive Hoare triples for entire programs.

We proceed as follows: We first define rules for Hoare triples for the statement/program language of C0 and see how we can use them to prove programs correct *manually*. Then, we look into how we can use these rules in a *mechanized* setting, i.e. how we can create an *automatic verifier*, a computer program that uses these rules to *automatically* prove programs correct.

To keep it simple, we focus on the initial definition of the C0 language in Section 6.2 and disregard pointers and blocks with local variables here.

7.4.1 Hoare Logic

Before we start, we need to make an important distinction. In Definition 7.1.7 and Definition 7.1.6 we have defined Hoare triples *semantically* which means that we used properties of the program execution of a C0 program (more specifically its termination behavior) to define what a Hoare triple is. In this section, we are interested in defining a *logic* which means that we define a set of rules over the *syntax* of a language (here C0) to derive facts (here Hoare triples) about a term in that language (here a C0 program). To make the difference clear, we write

$$\vDash \{P\}\ p\ \{Q\}$$

when we mean that program p is in fact totally/partially correct with respect to the specification (P, Q). This is the notion of a Hoare triple that we have introduced in Definition 7.1.7 and Definition 7.1.6. We write

$$\vdash \{P\}\ p\ \{Q\}$$

when we mean that we derived the Hoare triple with the proof rules we develop in this section. Of course, the proof rules we define are only useful if they are sound which means that the Hoare triples we can derive with them are *semantically* true which formally means that $\vdash \{P\}\ p\ \{Q\}$ entails $\vDash \{P\}\ p\ \{Q\}$. The rules we define in this section are sound but we do not prove it here because it requires a slightly more involved technical setup that is beyond the scope of this text.

In the following, we define Hoare triples first for the statements of C0 and then define how they can be lifted to Hoare triples of programs.

Weakening and Strengthening. An assertion B is *stronger* than an assertion C if $B \Longrightarrow C$. Correspondingly, C is *weaker* than B. Based on the definition of satisfiability (Definition 7.1.4), a weaker assertion is satisfied by more states. For example

$$x > 10 \text{ is stronger than } x > 3$$
$$x = 10 \land y = 5 \text{ is stronger than } y = 5$$

In a Hoare triple we can strengthen preconditions and weaken postconditions. Intuitively, this is true because of the following observation: If $P' \Longrightarrow P$, each state satisfying P' also satisfies P, so if $\vdash \{P\}\ s\ \{Q\}$, we also have $\vdash \{P'\}\ s\ \{Q\}$. Similarly, if $Q \Longrightarrow Q'$, each state that satisfies Q also satisfies Q'. Hence, if we have $\vdash \{P\}\ s\ \{Q\}$, we then also have $\vdash \{P\}\ s\ \{Q'\}$. This is reflected in the following rule which is often called **rule of consequence**:

$$[\text{Consequence}]\ \frac{P' \Rightarrow P \qquad \vdash \{P\}\ s\ \{Q\} \qquad Q \Rightarrow Q'}{\vdash \{P'\}\ s\ \{Q'\}}$$

Abort. Aborting the program means that it doesn't *properly* terminate (refer to Remark 6.3.7 and Definition 7.1.7). Hence, in *no* state does the program `abort();` terminate in a state that would satisfy any postcondition. Therefore, for any postcondition Q the Hoare triple of `abort();` is

$$[\text{Abort}]\ \frac{}{\vdash \{\mathit{false}\}\ \texttt{abort();}\ \{Q\}}$$

Assignment. Consider the assignment statement of C0 $x = e;$ and a post-condition Q. Now, let us consider what could be a valid *pre-condition* P that makes the Hoare triple $\{P\}\ x = e;\ \{Q\}$ true.

To this end, let us review the semantics of the assignment (see to Definition 6.3.4) in detail: First of all, the result of the R-evaluation of the expression e must be defined which is ensured by the predicate def e. Second, the effect of $x = e;$ is to evaluate e and assign its value to the variable x:

$$\langle x = e; \mid \sigma\rangle \longrightarrow \langle \varepsilon \mid \sigma'\rangle$$

So evaluating e on σ yields the same value as evaluating x on σ'. Hence, if σ' satisfies Q, σ will satisfy the assertion $Q[e/x]$ (read: e replaces x in Q). because e evaluates to the same value on σ as x does on σ'. That's just the purpose of the assignment. So the Hoare triple

$$\vdash \{\text{def } e \land Q[e/x]\}\ x = e;\ \{Q\}$$

is true and our proof system gets another axiom:

Definition 7.4.1 Hoare triple for the assignment statement.

$$[\text{Assign}]\ \frac{}{\vdash \{\text{def } e \wedge Q[e/x]\}\ x = e;\ \{Q\}}$$

◊

The def e part comes from the fact that in C0, the evaluation of expressions can get stuck, so the precondition needs to capture that because if the assignment gets stuck, it neither diverges nor terminates properly.

Let us check this on some examples:

Example 7.4.2

$$\frac{}{\vdash \{2 > 0\}\ \mathsf{x = 2;}\ \{\mathsf{x} > 0\}}$$

$$\frac{}{\vdash \{2 * \mathsf{y} = \mathsf{y}\}\ \mathsf{x = 2 * y;}\ \{\mathsf{x} = \mathsf{y}\}}$$

$$\frac{}{\vdash \{\mathsf{y} + 1 = 5\}\ \mathsf{x = y + 1;}\ \{\mathsf{x} = 5\}}$$

$$\frac{}{\vdash \{\mathsf{z} \neq 0 \wedge \mathsf{y} / \mathsf{z} = 5\}\ \mathsf{x = y / z;}\ \{\mathsf{x} = 5\}}$$

□

Block. The Hoare triple for a block just falls back to the Hoare triple of its constituent program (rule [Block]). The empty program ε does nothing so the Hoare triple $\{Q\}\ \varepsilon\ \{Q\}$ is valid. More interesting is the Hoare triple for the program that consists of a statement and a rest program.

In principle, the following rule says that if we can prove Hoare triples of the first statement and the remainder of the block in which the postcondition of the first statement is the precondition of the remainder, we can derive a Hoare triple of the entire block. Intuitively this is justified by the observation that the first Hoare triple yields that, if s_1 terminates, it terminates in a state that fulfills the precondition of the remainder of the block which then, by its Hoare triple, will, (if it terminates) terminate in a state that fulfills Q.

Definition 7.4.3 Hoare triples for the block statement and programs.

$$[\text{Seq}]\ \frac{\vdash \{P\}\ s\ \{Q\} \qquad \vdash \{Q\}\ p\ \{R\}}{\vdash \{P\}\ s\ p\ \{R\}}$$

$$[\text{Term}]\ \frac{}{\vdash \{Q\}\ \varepsilon\ \{Q\}}$$

$$[\text{Block}]\ \frac{\vdash \{P\}\ p\ \{Q\}}{\vdash \{P\}\ \{p\}\ \{Q\}}$$

Because sequences of statements (which form a program) are always terminated by ε, we define a convenience rule for a program that contains of two statements. This rule can be used to conveniently "process" the last two statements of a longer program without having to resort to the [Term] rule.

$$[\text{SeqS}]\ \frac{\vdash \{P\}\ s_1\ \{Q\} \qquad \vdash \{Q\}\ s_2\ \{R\}}{\vdash \{P\}\ s_1\ s_2\ \varepsilon\ \{R\}}$$

◊

Example 7.4.4 Let us consider the following program, that exchanges the contents of two variables:

$$\{\mathtt{t = x; x = y; y = t;}\}$$

When formulating the specification, we run into a small problem: We'd like to express in the postcondition that y equals the value that x had *at the beginning* of the program. However, the program overwrites x and we cannot relate to x initial value anymore. This can be solved by using **ghost variables** (see (7.3)) that capture the initial values of all variables in the program but are forbidden to be modified by the program itself.

So, the Hoare triple, we'd like to prove is:

$$\vdash \{\mathtt{Y = y} \land \mathtt{X = x}\}\ \{\mathtt{t = x; x = y; y = t;}\}\ \{\mathtt{Y = x} \land \mathtt{X = y}\}$$

For now, we know only the precondition and the postcondition because they are part of our proof goal. But the [Seq] rule requires that we come up with a condition R to split up the block Hoare triple into smaller ones. We have to *choose* R so that we can complete the proofs. Initially, it may not be clear how to pick the right R so that the proof goes through. In our simple example, we can however exploit the fact that the constituent statements solely are assignment statements: looking back at Definition 7.4.1 we see that the precondition of the Hoare triple of an assignment is actually *derived* from the postcondition. We can use this insight to derive all the Rs in our proof tree. To this end, let us sketch the proof tree first and then derive all the missing Rs using the assignment's Hoare triple.

$$[\text{Block}]\,\dfrac{[\text{Seq}]\,\dfrac{\vdash \{R_3\}\ \mathtt{t = x;}\ \{R_2\} \qquad [\text{SeqS}]\,\dfrac{\vdash \{R_2\}\ \mathtt{x = y;}\ \{R_1\} \qquad \vdash \{R_1\}\ \mathtt{y = t;}\ \{Q\}}{\vdash \{R_2\}\ \mathtt{x = y; y = t;}\ \{Q\}}}{\vdash \{\mathtt{Y = y} \land \mathtt{X = x}\}\ \mathtt{t = x; x = y; y = t;}\ \{\mathtt{Y = x} \land \mathtt{X = y}\}}}{\vdash \underbrace{\{\mathtt{Y = y} \land \mathtt{X = x}\}}_{=P}\ \{\mathtt{t = x; x = y; y = t;}\}\ \underbrace{\{\mathtt{Y = x} \land \mathtt{X = y}\}}_{=Q}}$$

Now, the assignment Hoare triple determines the Rs in the following way:

$$\begin{aligned} R_1 &= \mathtt{Y = x} \land \mathtt{X = t} \\ R_2 &= \mathtt{Y = y} \land \mathtt{X = t} \\ R_3 &= \mathtt{Y = y} \land \mathtt{X = x} \end{aligned}$$

We see that the assertions R_3 and P are *syntactically* the same which completes the proof. We come back to this kind of proof technique in Subsection 7.4.2 where we actually develop an algorithm to determine the missing invariants. □

If-Then-Else. A proof for the Hoare triple $\vdash \{P\}\ \mathtt{if}\,(e)\,s_1\,\mathtt{else}\,s_2\ \{Q\}$ can be obtained from proofs about the constituent statements. However, the preconditions of the Hoare triples of the then and else case can be strengthened by the conditional because we know that each case that s_1 is executed in also satisfies e (and similarly for s_2 and $\neg e$).

Definition 7.4.5 Hoare triple for if-then-else.

$$[\text{If}]\,\dfrac{\vdash \{e \land P\}\ s_1\ \{Q\} \qquad \vdash \{\neg e \land P\}\ s_2\ \{Q\}}{\vdash \{P\}\ \mathtt{if}\,(e)\,s_1\,\mathtt{else}\,s_2\ \{Q\}}$$

◊

Example 7.4.6 Let us consider a small program that computes the minimum two numbers:

$$\mathtt{if\,(x < y)\ r = x;\ else\ r = y;}$$

The Hoare triple we would like to prove is:

$$\vdash \{true\}\ \texttt{if (x < y) r = x; else r = y;}\ \{\texttt{r} \le \texttt{x} \wedge \texttt{r} \le \texttt{y} \wedge (\texttt{r} = \texttt{x} \vee \texttt{r} = \texttt{y})\}$$

By Definition 7.4.5, we get the following first step in our proof tree:

$$[\text{If}]\ \frac{\vdash \{x < y\}\ \texttt{r = x;}\ \{Q\} \qquad \vdash \{x \ge y\}\ \texttt{r = y;}\ \{Q\}}{\vdash \{true\}\ \texttt{if (x < y) r = x; else r = y;}\ \{\underbrace{\texttt{r} \le \texttt{x} \wedge \texttt{r} \le \texttt{y} \wedge (\texttt{r} = \texttt{x} \vee \texttt{r} = \texttt{y})}_{=Q}\}}$$

Now, we run into a small problem. Let's consider the triple for the then-part: using rule [Assign] (Definition 7.4.1), we can prove the triple

$$\frac{}{\vdash \{Q[\texttt{x}/\texttt{r}]\}\ \texttt{r = x;}\ \{Q\}}$$

but not

$$\frac{}{\vdash \{x < y\}\ \texttt{r = x;}\ \{Q\}}$$

Both preconditions are *syntactically* different ($Q[\texttt{x}/\texttt{r}] \ne x < y$) but logically equivalent ($Q[\texttt{x}/\texttt{r}] \equiv x < y$): they are satisfied by the exact same set of states. So, we cannot apply the rule [Assign] directly.

We can solve this problem by using the rule of consequence to generate an appropriate **verification condition**:

$$[\text{Consequence}]\ \frac{x < y \Rightarrow Q[\texttt{x}/\texttt{r}] \qquad \dfrac{}{\vdash \{Q[\texttt{x}/\texttt{r}]\}\ \texttt{r = x;}\ \{Q\}}\ [\text{Assign}]}{\vdash \{x < y\}\ \texttt{r = x;}\ \{Q\}}$$

Verification conditions are logical formulas that result from the [Consequence] rule and constitute additional premises that have to be satisfied in a Hoare-logic proof. We will come back to them in Subsection 7.4.2.

Coming back to the example above, the else case can be dealt with similarly and generates another verification condition:

$$[\text{Consequence}]\ \frac{x \ge y \Rightarrow Q[\texttt{y}/\texttt{r}] \qquad \dfrac{}{\vdash \{Q[\texttt{y}/\texttt{r}]\}\ \texttt{r = y;}\ \{Q\}}\ [\text{Assign}]}{\vdash \{x \ge y\}\ \texttt{r = y;}\ \{Q\}}$$

□

Loops. To define the proof rule for the while loop we resort to the concept of a **loop invariant**. A loop invariant is an assertion that holds true before and after the execution of a loop body. Hence, if some assertion I is a loop invariant, it will also hold before the loop was entered and after the loop was left (if it was left at all). This gives rise to the following proof rule:

$$[\text{While}]\ \frac{\vdash \{I \wedge e\}\ s\ \{I\}}{\vdash \{I\}\ \texttt{while}\ (e)\ s\ \{I \wedge \neg e\}}$$

The premise expresses that I must be a loop invariant. Note that we can strengthen the precondition because whenever s is executed, e also holds. In a similar way we know that as soon as we have left the loop, e will be false, hence we can stipulate the postcondition $I \wedge \neg e$.

Example 7.4.7 Let us consider an example here as well. We subtract `y` from `x` until `x` becomes less than `y`. At the same time, we count how many times we subtracted `y` in q. In the end, `q` is the quotient of `x` divided by `y` and `x` contains the remainder.

$$\dfrac{\dfrac{}{\vdash \{P\}\ \mathtt{q = 0;}\ \{I\}}\text{[Assign]} \qquad \mathsf{Tree}_W}{\vdash \{\underbrace{X = x}_{=P}\}\ \mathtt{q = 0;} \underbrace{\mathtt{while\,(x \geq y)\ \{x = x - y; q = q + 1;\}}}_{=:W}\ \underbrace{\{\underbrace{\mathtt{q \cdot y + x = X}}_{=I} \wedge \mathtt{x < y}\}}_{=Q}}\text{[SeqS]}$$

For better readability, we split off a part of the proof tree into a separate part designated by Tree_W.

$$\dfrac{\dfrac{\dfrac{\dfrac{I \wedge \mathtt{x \geq y} \Rightarrow R[\mathtt{x - y/x}] \qquad \dfrac{}{\vdash \{R[\mathtt{x - y/x}]\}\ \mathtt{x = x - y;}\ \{R\}}\text{[Assign]}}{\vdash \{I \wedge \mathtt{x \geq y}\}\ \mathtt{x = x - y;}\ \{R\}}\text{[Csq]} \qquad \dfrac{}{\vdash \{R\}\ \mathtt{q = q + 1;}\ \{I\}}\text{[Assign]}}{\vdash \{I \wedge \mathtt{x \geq y}\}\ \mathtt{x = x - y; q = q + 1;}\ \{I\}}\text{[SeqS]}}{\vdash \{I \wedge \mathtt{x \geq y}\}\ \mathtt{\{x = x - y; q = q + 1;\}}\ \{I\}}\text{[Block]}}{\vdash \{I\}\ W\ \{I \wedge \mathtt{x < y}\}}\text{[While]}$$

For the correctness proof, we need to remember the original value of x with a ghost variable. The invariant $\mathtt{q \cdot y + x = X}$ is conveniently also the postcondition of the program together with the fact that the remainder is smaller than the divisor. Therefore, we do not need the consequence rule for the while loop.

Let us go through the proof from right to left. After the loop, the condition is false. Therefore, we know $\mathtt{x < y}$. It remains to show that the invariant $\mathtt{q \cdot y + x = X}$ is true before the loop was entered and in each iteration of the loop. We can do this by using the [While] rule.

Before and after each iteration of the loop, `q` contains the count of how many times we subtracted `y` from `x`. In the loop, we subtract `y` from `x` one additional time and increase `q` by one. By doing this, we invalidate that invariant for a moment in the loop body but restore it afterward before reaching the end of the loop body. The intermediate condition R is true at the point before `q` is increased but after `y` is subtracted from `x`.

$$R = I[\mathtt{q + 1/q}] = (\mathtt{q \cdot y + x + y = X})$$

By applying the [Assign] rule to R, we get the back to the invariant:

$$I = R[\mathtt{x - y/x}] = (\mathtt{q \cdot y + x - y + y = X}) = (\mathtt{q \cdot y + x = X})$$

Lastly, it remains to show that the invariant was satisfied before the loop was entered. This is done by using the [Assign] rule for the first assignment.

$$P = I[\mathtt{0/q}] = (\mathtt{0 \cdot y + x = X}) = (\mathtt{x = X})$$

□

Checkpoint 7.4.8 Prove that the remainder is non-negative for a non-negative x.
Do you need the pre-condition $\mathtt{0 < y}$? Why or why not?

7.4.2 Mechanizing Verification

We have seen in the examples in Subsection 7.4.1 that some rules require the verifier to “guess” assertions to complete the proofs. This was the case for the rules for the sequence, if-then-else, and the while loop. In the examples, we also made the experience that sometimes these free assertions were determined by other rules: for example, the assignment rule essentially defines the precondition based on its postcondition. This helped us to determine the “free” assertions of the sequence rule in Example 7.4.4.

In this section, we introduce two functions

$$\begin{aligned} \mathrm{pc} &: (\mathit{Stmt} \cup \mathit{Prg}) \longrightarrow \mathit{Assert} \longrightarrow \mathit{Assert} \\ \mathrm{vc} &: (\mathit{Stmt} \cup \mathit{Prg}) \longrightarrow \mathit{Assert} \longrightarrow \mathcal{P}(\mathit{Assert}) \end{aligned}$$

that determine, given a postcondition Q, for each statement or program, a precondition and a set of verification conditions such that

$$\frac{\mathrm{vc}\, q\, Q}{\vdash \{\mathrm{pc}\, q\, Q\}\ q\ \{Q\}}$$

where q is either a statement or a program. We can use such rules together with the consequence rule to prove a program correct with respect to a specification (P, Q) in the following way:

$$\frac{P \Rightarrow \mathrm{pc}\, q\, Q \qquad \dfrac{\mathrm{vc}\, q\, Q}{\vdash \{\mathrm{pc}\, q\, Q\}\ q\ \{Q\}}}{\vdash \{P\}\ q\ \{Q\}}$$

The idea behind this is that the function pc determines a precondition that describes the states under which an execution of q properly terminates (if it terminates at all) in a state that satisfies a given postcondition Q. One can indeed show that except for the while loop, the function pc gives the **weakest precondition**, i.e. the precondition that describes *exactly those states* for which the program terminates properly in a state that satisfies the postcondition Q.

The function vc collects verification conditions that are necessary to justify $\vdash \{\mathrm{pc}\, q\, Q\}\ q\ \{Q\}$. In doing so, we essentially defer the proof of the program's correctness to a **theorem prover**, a tool that can check if a verification condition is valid or not.[16]

Definition 7.4.9 Preconditions and Verification Conditions.

$$\begin{aligned} \mathrm{pc}\, [\![\varepsilon]\!]\, Q &= Q \\ \mathrm{pc}\, [\![s\ p]\!]\, Q &= \mathrm{pc}\ s\ (\mathrm{pc}\ p\ Q) \\ \mathrm{pc}\, [\![\mathtt{x} = e\mathtt{;}]\!]\, Q &= Q[e/\mathtt{x}] \\ \mathrm{pc}\, [\![\{p\}]\!]\, Q &= \mathrm{pc}\ p\ Q \\ \mathrm{pc}\, [\![\mathtt{if}\ (e)\ s_1\ \mathtt{else}\ s_2]\!]\, Q &= (e \wedge \mathrm{pc}\ s_1\ Q) \vee (\neg e \wedge \mathrm{pc}\ s_2\ Q) \\ \mathrm{pc}\, [\![\mathtt{while}\ (e)\ \mathtt{\_Inv}(I)\ s]\!]\, Q &= I \end{aligned}$$

$$\begin{aligned} \mathrm{vc} [\![\varepsilon]\!]\, Q &= \{\} \\ \mathrm{vc} [\![s\ p]\!]\, Q &= \mathrm{vc}\, s\ (\mathrm{pc}\ p\ Q) \cup \mathrm{vc}\ p\ Q \\ \mathrm{vc} [\![\mathtt{x} = e\mathtt{;}]\!]\, Q &= \{\} \\ \mathrm{vc} [\![\{p\}]\!]\, Q &= \mathrm{vc}\, p\, Q \\ \mathrm{vc} [\![\mathtt{if}\ (e)\ s_1\ \mathtt{else}\ s_2]\!]\, Q &= (\mathrm{vc}\, s_1\ Q) \cup (\mathrm{vc}\, s_2\ Q) \\ \mathrm{vc} [\![\mathtt{while}\ (e)\ \mathtt{\_Inv}(I)\ s]\!]\, Q &= \{I \wedge e \Rightarrow \mathrm{pc}\, s\, I, \neg e \wedge I \Rightarrow Q\} \cup (\mathrm{vc}\, s\, I) \end{aligned}$$

◊

The while rule does not so nicely fit into this schema because the invariant of the while loop cannot be automatically inferred but has to be annotated by the programmer, hence the syntax `_Inv`(I). Especially, the invariant cannot be automatically

[16]Note that the decidability of verification conditions strongly depends on the theories the conditions use. Some of the practically most relevant theories (e.g. arithmetic on natural numbers including multiplication) are not decidable, which means that the theorem prover might not terminate.

deduced from the postcondition and in the definition of pc for while, the postcondition Q is simply ignored. Instead, the while case relies on several verification conditions that "connect" the while loop to the rest of the program.

The following theorems state that Definition 7.4.9 is sound.

Theorem 7.4.10 *For each statement s, the following rule is sound.*

$$\frac{\mathrm{vc}\, s\, Q}{\vdash \{\mathrm{pc}\, s\, Q\}\ s\ \{Q\}}$$

Proof. Sketch. By induction over s. The assignment and empty block case are trivial; just put in the definition of pc and vc into the rule of the theorem and we get the empty block and assignment rules from Subsection 7.4.1. We'll show the if-then-else and while case here.

- if-then-else: $s = \texttt{if}\,(e)\, s_1\, \texttt{else}\, s_2$. By induction, we have that the rules for s_1 and s_2 are sound. Hence, we can derive the hoare triple in question:

$$[\text{Conseq}]\ \ [\text{If}]\ \dfrac{\dfrac{\dfrac{\mathrm{vc}\, s_1\, Q}{\vdash \{\mathrm{pc}\, s_1\, Q\}\ s_1\ \{Q\}}}{\vdash \{e \land \mathrm{pc}\, s\, Q\}\ s_1\ \{Q\}} \qquad \dfrac{\dfrac{\mathrm{vc}\, s_2\, Q}{\vdash \{\mathrm{pc}\, s_2\, Q\}\ s_2\ \{Q\}}}{\vdash \{\neg e \land \mathrm{pc}\, s\, Q\}\ s_2\ \{Q\}}\ [\text{Conseq}]}{\vdash \{\mathrm{pc}\, s\, Q\}\ s\ \{Q\}}$$

  Using the consequence rule, we strengthen the preconditions of s_1 and s_2 accordingly. $\mathrm{pc}\, s\, Q$ is now defined such that $e \land \mathrm{pc}\, s\, Q \implies \mathrm{pc}\, s_1\, Q$ and $\neg e \land \mathrm{pc}\, s\, Q \implies \mathrm{pc}\, s_2\, Q$. The verification conditions that are needed to justify this derivation are just the ones of s_1 and s_2, so $\mathrm{vc}\, s\, Q = (\mathrm{vc}\, s_1\, Q) \cup (\mathrm{vc}\, s_2\, Q)$

- The while loop $\texttt{while}\,(e)\,\texttt{\_Inv}(I)\, s$: We need to prove that $\vdash \{\mathrm{pc}[\![\texttt{while}\,(e)\,\texttt{\_Inv}(I)\, s]\!]\, Q\}\ \texttt{while}\,(e)\,\texttt{\_Inv}(I)\, s\ \{Q\}$ with $\mathrm{pc}\, s\, Q = I$. By induction, we have that the rule

$$\frac{\mathrm{vc}\, s\, I}{\vdash \{\mathrm{pc}\, s\, I\}\ s\ \{I\}}$$

  is sound. We use this to derive our proof goal using the following derivation:

$$[\text{Conseq}]\ \dfrac{[\text{While}]\ \dfrac{[\text{Conseq}]\ \dfrac{I \land e \implies \mathrm{pc}\, s\, I \qquad \dfrac{\mathrm{vc}\, s\, I}{\vdash \{\mathrm{pc}\, s\, I\}\ s\ \{I\}}}{\vdash \{I \land e\}\ s\ \{I\}}}{\vdash \{I\}\ \texttt{while}\,(e)\,\texttt{\_Inv}(I)\, s\ \{\neg e \land I\}} \qquad \neg e \land I \implies Q}{\vdash \{I\}\ \texttt{while}\,(e)\,\texttt{\_Inv}(I)\, s\ \{Q\}}$$

  In that derivation, we used the consequence rule twice and generate two additional verification conditions: $I \land e \implies \mathrm{pc}\, s\, I$ and $\neg e \land I \implies Q$ which is reflected in the definition of vc.

■

Theorem 7.4.11 *For each program p, the following rule is sound.*

$$\frac{\mathrm{vc}\, p\, Q}{\vdash \{\mathrm{pc}\, p\, Q\}\ p\ \{Q\}}$$

Proof. Exercise. ■

Example 7.4.12 Consider the division program from Example 7.4.7.

$$\texttt{q} = \texttt{0}; \texttt{while}\,(\texttt{x} \geq \texttt{y})\,\texttt{\_Inv}(\texttt{q} * \texttt{y} + \texttt{x} = \texttt{X})\ \underbrace{\{\texttt{x} = \texttt{x} - \texttt{y}; \texttt{q} = \texttt{q} + 1;\}}_{s}$$

which we want to prove correct with respect to the specification

$$\begin{aligned} P &= \texttt{x} = \texttt{X} \\ Q &= \texttt{q} * \texttt{y} + \texttt{x} = \texttt{X} \land \texttt{x} < \texttt{y} \end{aligned}$$

using the loop invariant

$$I = \texttt{q} * \texttt{y} + \texttt{x} = \texttt{X}$$

Using Definition 7.4.9, we have:

$$\begin{aligned} \text{pc}[\![\texttt{while} \ldots]\!]\, Q &= I \\ \text{pc}[\![\texttt{q} = \texttt{0}; \ldots]\!]\, Q &= \text{pc}[\![\texttt{q} = \texttt{0};]\!]\, (\text{pc}[\![\texttt{while} \ldots]\!]\, Q) \\ &= \text{pc}[\![\texttt{q} = \texttt{0};]\!]\, I = (\texttt{0} * \texttt{y} + \texttt{x} = \texttt{X}) \end{aligned}$$

and

$$\begin{aligned} \text{vc}[\![\texttt{q} = \texttt{0}; \ldots]\!]\, Q &= \text{vc}[\![\texttt{q} = \texttt{0};]\!]\, (\text{pc}[\![\texttt{while} \ldots]\!]\, Q) \cup \text{vc}[\![\texttt{while} \ldots]\!]\, Q \\ &= \text{vc}[\![\texttt{while} \ldots]\!]\, Q \\ \text{vc}[\![\texttt{while} \ldots]\!]\, Q &= (\text{vc}\, s\, I) \cup \{\texttt{x} < \texttt{y} \land I \Rightarrow Q, I \land \texttt{x} \geq \texttt{y} \Rightarrow \text{pc}\, s\, I\} \\ \text{vc}\, s\, I &= \emptyset \\ \text{pc}\, s\, I &= [(\texttt{q} + 1) * \texttt{y} + (\texttt{x} - \texttt{y}) = \texttt{X}] \end{aligned}$$

So in summary, we have:

$$\begin{aligned} \text{pc}[\![\texttt{q} = \texttt{0}; \ldots]\!]\, Q &= \texttt{0} * \texttt{y} + \texttt{x} = X \\ \text{vc}[\![\texttt{q} = \texttt{0}; \ldots]\!]\, Q &= \{\texttt{x} < \texttt{y} \land I \Rightarrow Q, I \land \texttt{x} \geq \texttt{y} \Rightarrow \text{pc}\, s\, I\} \end{aligned}$$

Coming back to (7.4.2), all verification conditions are:

1. $P \Rightarrow \texttt{0} * \texttt{y} + \texttt{x} = X$
2. $\texttt{x} < \texttt{y} \land I \Rightarrow Q$
3. $I \land \texttt{x} \geq \texttt{y} \Rightarrow (\texttt{q} + 1) * \texttt{y} + (\texttt{x} - \texttt{y}) = \texttt{X}$

They can now be given to a theorem solver. If the solver reports that they are valid, the program is partially correct with respect to the given specification. □

Assert and Assume. Let us define two further language elements that will allow us to specify pre- and postconditions directly in the program. Assert is defined as

$$\texttt{assert}(e) = \texttt{if}\,(e)\,\{\}\,\texttt{else}\,\texttt{abort}();$$

and therefore

$$\text{pc}[\![\texttt{assert}(e)]\!]\, Q = e \land Q \qquad \text{vc}[\![\texttt{assert}(e)]\!]\, Q = \{\}$$

We use the statement $\texttt{assume}(a)$ to specify pre-conditions or give additional constraints. In a more realistic setting, one could for example provide additional information on inputs, like maybe some input is always in a specific range, etc.

Assume is directly defined through a while loop that makes the program diverge if the condition e is not fulfilled. This leads to a loop invariant that states that $\neg e$ or Q holds (or written differently: $e \Rightarrow Q$). The Q part comes from the fact that if the loop terminates (in this case equivalent to not being entered), we want the postcondition Q is satisfied.

$$\texttt{assume}(e) \coloneqq \texttt{while}\,(\neg e)\;\texttt{\_Inv}(e \Rightarrow Q)\,\{\}$$

and with the loop invariant $e \Rightarrow Q$, if follows that

$$\text{pc}[\![\texttt{assume}(e)]\!]\,Q = e \Rightarrow Q \qquad \text{vc}[\![\texttt{assume}(e)]\!]\,Q = \emptyset$$

Lemma 7.4.13 $\{\mathit{true}\}\ \texttt{assume}(P); s; \texttt{assert}(Q);\ \{\mathit{true}\}$ *iff* $\{P\}\ s\ \{Q\}$

Proof. Exercise. Investigate the proof tree for the left side. ■

7.5 Summary and Further Reading

TODO

Chapter 8

Object-Oriented Programming with Java

Java is in its syntax very close to C. We find most statements and operators that we know from C also in Java. In this chapter, we start by briefly discussing the most important differences between Java and C. Then, we will go into more detail regarding the object-oriented language features in Java.

8.1 Compilation, Main Method, Packages

A Java program consists of one or more files with the file ending `.java`. Every `.java` file declares a **class**. This class must have the same name as the file. For example, a file `X.java` needs to contain a declaration for a class `X`:

```java
public class X {
}
```

Aside: Strictly speaking, more than one class can be declared in a single `.java` file. This is however a rarely used feature and comes with restrictions: At most one class in a `.java` file may be declared as `public`.

The Java compiler `javac` translates a file `X.java` into the **class file** `X.class`. This file corresponds to the `.o` object files that we know from C. It contains information about the class and the so-called **Java bytecode** for the functions of the class. Java bytecode is an abstract machine language that is executed by the Java runtime environment. Java programs are started from the command line with a call to the Java runtime environment: `java X`. For this to work, there needs to be a class file `X.class` that contains a function with the name `main`. Our first Java program is therefore:

```java
public class X {
    public static void main(String[] args) {
        System.out.println("Hello␣World");
    }
}
```

We compile it to a class file with

```
$ javac X.java
```

which we execute with

```
$ java X
```

Java recognizes a function `main` only as the program's starting point if it has the return type `void` and a parameter of the array type `String[]`. The `main` function further needs to be decorated with the keywords `public` and `static`. The parameter `args` of `main` is a sequence of character strings. It is used to pass the provided command line parameters to the program. For instance, the following program prints its command line parameters, each in a separate line:

```java
public class ArgPrinter {
    public static void main(String[] args) {
        for (String s : args)
            System.out.println(s);
    }
}
```

It operates as follows:

```
$ java ArgPrinter these are five arguments now
these
are
five
arguments
now
```

In contrast to C, the name of the executable is not included in the argument sequence.

8.1.1 Multiple Classes

Realistic Java programs consist of more than one class (and therefore also multiple files). Each class defines its own visibility scope or **namespace**. The identifiers that are declared in a class (like, for instance, `main` above), are by default only visible inside the class. By **qualification**, we can still use them in other scopes:

```java
// Y.java
public class Y {
    static void sayHello() {
        System.out.println("Hello");
    }
}

// X.java
public class X {
    public static void main(String[] args) {
        Y.sayHello();
    }
}
```

In this example, the identifier `Y.sayHello` is qualified with the name of the class where it is declared. Java provides ways to restrict the use of qualification to manage the accessibility of identifiers. Section 8.5 provides more details on this topic.

8.1.2 Packages

The many classes of a typical Java project can be structured in packages. We express that the class `X` is part of the package `p` with a package statement as follows:

```java
package p;

public class X {
}
```

The package structure is implicitly linked to the directory structure of the `.java` files. In the above example, the Java compiler will expect that the file `X.java` is in the subdirectory `p`. Classes from other packages can be accessed either via qualification with the package name or by importing the package. For instance, consider the class `Y` in the package `p1`:

```java
// p1/Y.java
package p1;

public class Y {
    static void sayHello() {
        System.out.println("Hello");
    }
}
```

If we want to refer to `Y` from within a class `X` in the package `p2`, we can do so as follows, using qualification:

```java
// p2/X.java
package p2;
public class X {
    public static void main(String[] args) {
        p1.Y.sayHello();
    }
}
```

Or we can use a package import:

```java
// p2/X.java
package p2;

import p1.Y;

public class X {
    public static void main(String[] args) {
        Y.sayHello();
    }
}
```

8.2 Types

In Java, we distinguish **primitive types** and **reference types**. The reference types are further categorized into **classes** and **array types**. The primitive types are very similar to types we know from C. However, their size and arithmetic is exhaustively defined in Java.

Table 8.2.1 Primitive types in Java

Type	Description	Value Range	Default Value
`byte`	8-bit integer	$-128 \dots 127$	`0`
`short`	16-bit integer	$-32768 \dots 32767$	`0`
`int`	32-bit integer	$-2^{31} \dots 2^{31}-1$	`0`
`long`	64-bit integer	$-2^{63} \dots 2^{63}-1$	`0L`
`char`	16-bit Unicode character	`'\u0000'` … `'\uFFFF'`	`'\u0000'`
`float`	32-bit IEEE 754 Floating Point		`0.0f`
`double`	64-bit IEEE 754 Floating Point		`0.0`
`boolean`	truth values	`false`, `true`	`false`

The primitive types can be implicitly converted according to the order in Figure 8.2.2. This order results from the subset relation of the value ranges given in Table 8.2.1. `char` is the only *unsigned* data type in java and therefore neither subset nor superset of `short` and `byte`. The type `boolean` cannot be converted implicitly to another type and no type is implicitly converted to `boolean`.

$$\begin{tikzcd}
\texttt{byte} \arrow[r] & \texttt{short} \arrow[r] & \texttt{int} \arrow[r] & \texttt{long} \arrow[r] & \texttt{float} \arrow[r] & \texttt{double} \\
\texttt{boolean} & \texttt{char} \arrow[ur] & & & &
\end{tikzcd}$$

Figure 8.2.2 Implicit type conversion between primitive types

For other conversions (excluding conversions to `boolean`), we can use explicit type casts: The expression `(T)e` converts the subexpression `e` to the type `T`. Such explicit casts can lose precision. For instance, `(byte)1024` is equal to `0`.

Example 8.2.3 Implicit and Explicit Type Conversions.

```java
short s = 1;
byte  b = 2;
int   i = 3;
i = s; // ok
b = s; // not ok, as short is not implicitly
       // convertible to byte
b = (byte)s; // ok, as it is an explicit cast
```

□

8.3 Reference Types

The container of a variable with a reference type always contains the reference to an object or the value `null`. In contrast to C, Java has no references to already freed objects that would cause undefined behavior if dereferenced. This is a result of Java's memory management strategy: Manually freeing memory like in C is not possible. As long as an object is accessible via a reference in the current program state (transitively), it is alive. No longer used memory is deallocated in Java via **garbage collection**. The Java runtime environment therefore regularly checks for objects that are no longer reachable via available references and frees their memory.

Aside: In Java, memory addresses are called "reference".

In contrast to C, it is impossible to declare variables with the type of a class or an array without indirection. Variables that are declared with such a type always implicitly contain reference types. The actual objects and arrays are always allocated on the heap with a `new` expression (see next section).

8.3.1 Arrays

A variable with an array type contains a reference to a container with the length of the array and its elements. An array of values of the type `T` can be allocated with the expression `new T[e]`, where `e` is an expression describing the length of the array. The `new` operator provides a reference to the newly create array. The length of an array is immutable: After being passed to the new operator, the length cannot change during the array's life time.

Consider the following example:

```
int[] a = new int[10];
int[] b = a;
a[2] = 42;
// print 42
System.out.println(b[2]);
```

Here, an array for 10 `int` elements is created. In line 2, the array is not copied, only a reference to it. The length of an array `a` can be determined with `a.length`:

```
// print 10
System.out.println(a.length);
```

Java also supports multi-dimensional arrays, for example:

```
int[][] arr    = new int[4][4];
int[]   subarr = arr[2];
```

In contrast to C, a multi-dimensional array is however not just a single container with space for all elements of the multi-dimensional array. Instead, it is an array of arrays. Since array types are reference types, every element of a multi-dimensional array is a reference to another (lower-dimensional) array. In the above example, `arr[2]` has the type `int[]`. The array `arr` contains references to arrays of the type `int`. Figure 8.3.1 shows the containers resulting from the above lines of code.

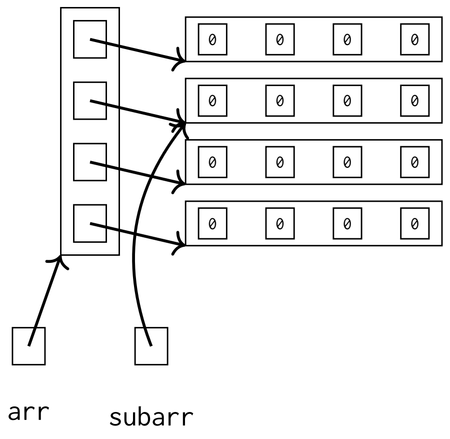


Figure 8.3.1 Multi-dimensional arrays in Java

8.4 Classes

In larger programs, we very commonly work with *compound data types* (called `struct` in C). A compound data type is the cartesian product of several component types. A compound data type's value is therefore a tuple of several component values. Consider the following struct in C, which is intended to represent a point in the two-dimensional plane:

```c
typedef struct {
  double x, y;
} vec2_t;
```

Open in Browser

Since modifications and computations on structs usually involve more than one struct field, it is common to implement them in functions:

```c
void vec2_translate(vec2_t *p, double dx, double dy) {
  p->x += dx;
  p->y += dy;
}

void vec2_length(vec2_t const *p) {
  return sqrt(p->x * p->x + p->y * p->y);
}
```

These functions share a common feature: Their first parameter is a pointer to a struct (i.e. the value of a compound data type). In object-oriented programming languages like Java, such functions are directly attached to the compound data type. In Java, we would write:

```java
class Vec2 {
  double x, y;

  void translate(double dx, double dy) {
    this.x += dx;
    this.y += dy;
  }

  double length() {
    return Math.sqrt(this.x * this.x + this.y * this.y);
  }
}
```

The object-oriented programming field has its own nomenclature: Compound data types are called **classes** and a value of a class is called an **object** or an **instance**. Functions that are connected to classes are **methods** and, as we have previously seen, pointers are called **references**. Every method has an implicit parameter `this`: It is automatically declared by the compiler and does not need to be declared by the programmer. `this` is a reference to an object of the class in which the method is declared. In the above example, the type of `this` is therefore `Vec2`. If identifiers that are declared in the class are not overshadowed by local variables or parameters of the method, the qualification `this.` can be omitted. This means that we can also define the `length` method of `Vec2` also as follows:

```java
class Vec2 {
  ...
  double length() {
    return Math.sqrt(x * x + y * y);
  }
}
```

8.4.1 Constructors

A class needs to provide one or more **constructors** to initialize freshly created instances of the class. A constructor has the same name as its class and no return type. For our example, it is sensible to add a constructor that initializes a `Vec2` object with an x- and a y-coordinate:

```java
class Vec2 {
  double x, y;

  public Vec2(double x, double y) {
    this.x = x;
    this.y = y;
  }
  ...
}
```

All member fields of an object that are not explicitly initialized by the constructor are initialized with their **default value** in Java (see Table 8.2.1).

A special case is the **default constructor**, which is implicitly declared in Java if no other constructor is declared in a class. It behaves as the following declaration:

```java
public class Vec2 {
  public Vec2() {
  }
}
```

We can create new objects with `new` and one of the constructors of the class:

```java
public class Test {
  public static void main(String[] args) {
    Vec2 a = new Vec2(1.0, 0.0);
    System.out.println(a.length());
  }
}
```

The value of the expression `new Vec2(1.0, 0.0)` has the type `Vec2` and is a reference to a newly created object of the class `Vec2`. Like in C, the lifetime of the container associated with the variable `a` is limited to the surrounding block. The referenced object however stays alive as long as the program state contains references to it. Once no living container contains a reference to the object anymore, it is deallocated by the garbage collection.

To call a method, the reference of the corresponding object does not occur explicitly in the parameter list, but it is prepended as qualification to the method name with a separating dot. When the call is executed, this reference is copied to the implicitly declared `this` container of the method.

Example 8.4.1 Consider the call `a.length()` in the above example. Although no expression is in the parentheses, the method `length` receives an argument: the value of the expression `a` (a reference to an object). Compare this to the C version `vec2_length` of this method, which explicitly takes a parameter. □

Remark 8.4.2 The reason for this special notation is that this first parameter plays a special role for choosing the method that is called, as we will see in Section 8.7.

8.5 Encapsulation

Object-oriented programming languages like Java provide means to enforce **encapsulation**, that is to hide the internals of a data structure. Our `Vec2` example from the previous section gives a reason why we would want to hide the internal workings of a data structure. Let us consider another class, where `Vec2` objects are used:

```java
public class Rectangle {
  Vec2 upperLeft, lowerRight;
  ...
  double area() {
    double dx = lowerRight.x - upperLeft.x;
    double dy = lowerRight.y - upperLeft.y;
    return Math.abs(dx * dy);
  }
}
```

The class `Rectangle` appears to model a rectangle and has two fields, `upperLeft` and `lowerRight`, that store the upper left and the lower right corners of the rectangle. The method `area` computes the rectangle's enclosed area.

At the moment, `Vec2` uses cartesian coordinates. The implementation of `area` in `Rectangle` explicitly relies on this choice: it accesses the `x` and `y` members of `Vec2`. But what if the author of the `Vec2` class wants to adjust the internal representation of a point, for instance by using polar coordinates instead? Then, all code that directly refers to the member fields of `Vec2` would break. In fact, by referring to the fields of `Vec2`, code of other classes makes itself dependent on the *internal* implementation details of `Vec2`.

To prevent such a dependency and to *decouple* the individual classes of a system, we *encapsulate* classes by **hiding** internal information of a class from the rest of the system.

```java
public class Vec2 {
  private double x, y;

  public double getX() { return x; }
  public double getY() { return y; }
  ...
}
```

In Java, every declaration of a class, a field, or a method can be prepended with one of the keywords `private`, `protected`, and `public`. These keywords restrict the visibility of the declared identifiers. Method and field identifiers that are declared `private` can only be used in the class that contains the declaration. This would prohibit access to the `x` and `y` fields of the `Vec2` class in methods of the class `Rectangle`. `public` signifies an unrestricted visibility. Table 8.5.1 summarizes all visibility modifiers of Java.

Table 8.5.1 Visibility modifiers in Java

	same class	same package	subclass	otherwise
`public`	✓	✓	✓	✓
`protected`	✓	✓	✓	—
none	✓	✓	—	—
`private`	✓	—	—	—

The data represented by an object are then accessed via **accessor methods** (commonly called "getters" and "setters"). They allow us to change the internal representation of the class without the need to adjust other classes. The accessor methods always provide the same view on an object "to the outside". If we now modify `Vec2` to use polar coordinates, only the accessor methods `getX` and `getY` need to be adjusted. Furthermore, there is benefit in using the accessors within the declaring class as well. In our example, we then do not need to adjust other methods whose implementation uses cartesian coordinates:

```java
public class Vec2 {
  private double r, phi;

  public double getX() { return r * Math.cos(phi); }
  ...
  public void setX(double x) {
    phi = Math.atan2(getY(), x);
    r   = x / Math.cos(phi);
  }
  ...

  public void translate(double dx, double dy) {
    setX(getX() + dx);
    setY(getY() + dy);
  }

  public double length() {
    return r;
  }
}
```

Ensuring Class Invariants. Encapsulation also helps to ensure that invariants among the fields of a class (so-called **class invariants**) are preserved. Consider the following class, which models fractions:

```java
public class Fraction {
  private int numerator, denominator;
  ...
  public setDenominator(int d) {
    assert d > 0 : "Denominator␣must␣be␣>␣0";
    this.denominator = d;
  }

  public boolean isPositive() {
    return numerator >= 0;
  }
}
```

The programmer appears to base their implementation of the class `Fraction` on the fact that the denominator is always greater than zero. Thus, "`d` > 0" is an invariant for all `Fraction` objects. If this invariant was violated, the code of the class would not work correctly. For instance, the method `isPositive` would yield an incorrect result for `denominator` < 0 and `numerator` < 0. With a setter method, we can ensure that the object never reaches an invalid state by raising a suitable exception when the invariant would be violated.

Encapsulating Constructors. Sometimes, we also want to encapsulate object construction and forbid other classes' code direct access to the constructor(s) of a class. In Java, this can be achieved by setting the constructor `private`. Object construction has then to be performed by a method. Typically, in a class with private constructors there is a `static` **factory method** that creates new objects of the class using the private constructor, like in the following example:

```java
public class Vec2 {
  private double x, y;

  private Vec2(double x, double y) {
    this.x = x;
    this.y = y;
  }

  public static Vec2 cartesian(double x, double y) {
    return new Vec2(x, y);
  }

  public static Vec2 polar(double r, double phi) {
    return new Vec2(r * Math.cos(phi), r * Math.sin(phi));
  }
}
```

Here, the methods `cartesian` and `polar` serve the purpose of constructing a `Vec2` in different ways. `polar` transforms the polar coordinates into cartesian ones and then calls the constructor while `cartesian` just passes its arguments through. Both methods also carry a sensible name that makes it possible for the programmer to understand how the parameters of the object construction is interpreted (cartesian vs polar).

Other very common application of private constructors are:

Controlling the number of objects of a an *immutable* class (see Section 8.6) For example, nothing prevents us to create multiple distinct objects that all represent the `Vec2` (1, 0). Sometimes one wants to make sure that there is only *one* object for a specific value. In that case, a factory method could look up in a map if an object was already created for a specific set of parameters and return this object instead of creating a new one.
An extreme variant of this case is the **singleton** where one wants to permit only one object per class.

Delegating the construction of the object Sometimes, it turns out that for specific combinations of parameters to the constructor, an object of another (sub-) class is better suited. In this case, a factory method can select which object to create.

8.6 References, Aliasing, and Immutable Objects

In Java, variables whose type is a class are always *references* to the corresponding object. Consider the following example:

```java
public class Foo {
  int x;
  int y;

  public static void main(String[] args) {
    Foo a = new Foo();
    a.x = 3;
    // from here on, a and b alias
    Foo b = a;
    b.x = 5;
    System.out.println(a.x);  // prints 5
    System.out.println(b.x);  // prints 5
    // from here on, a and b no longer alias
    b = new Foo();
    b.x = 3;
    System.out.println(a.x);  // prints 5
    System.out.println(b.x);  // prints 3
  }
}
```

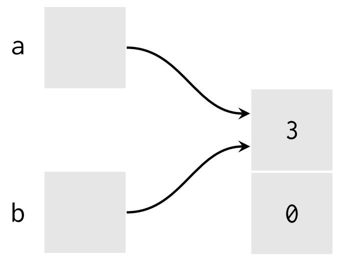


Figure 8.6.1 Visualization of the aliases `a` and `b` from the example code in memory.

When, at some point in time, an object has two references *a* and *b* referring to it, we say that *a* and *b* **alias** (or that they are **aliases**). In the above example, `a` and `b` are aliases of the same object between the corresponding comments.

8.6.1 Aliases

A benefit of using references is that they can be implemented with a compact memory footprint. The compiler typically translates them to an address to main memory, where the object's fields are located. For copying (as it also implicitly happens at a method call), only an address needs to be copied, rather than the entire object. However, multiple references to the same object at the same time can lead to program errors that are difficult to find. The problem is that *mutating* objects through aliases often opposes modularity: If we, for example, write a method that modifies an object *O*, a caller of the method needs to be aware of the fact that there may be other objects that refer to *O* and whether they can "handle" a potentially modified object. This awareness may require knowledge about other parts of the system and therefore is often not *modular* because the program's correctness then depends on the correct interplay of multiple objects (of potentially different classes).

8.6.2 Immutable Classes

One way to avoid errors caused by aliasing is to make classes immutable. This ensures that the values of the object's fields cannot change after the object is constructed. Consequently, aliasing references cannot modify the objects anymore. Copying the reference to an immutable object is therefore conceptually equivalent to copying it by value. Java supports enforcing immutability with the keyword `final`:

```java
public class Vec2 {
  private final double r, phi;

  public Vec2(double r, double phi) {
    this.r   = r;
    this.phi = phi;
  }
  ...
}
```

`final` fields of a class must receive a value in the constructor once. The Java compiler rejects any code that would modify a `final` field of an object at compile time. However, this immutability is also enforced for methods of the class, like the `translate` method in our example. For immutable classes, such a method needs to instead return a new, modified object rather than modifying the existing one:

```java
public class Vec2 {
  ...
  public Vec2 translate(double dx, double dy) {
    return cartesian(getX() + dx, getY() + dy);
  }

  public static Vec2 cartesian(double x, double y) {
    double phi = Math.atan2(y, x);
    double r   = x / Math.cos(phi);
    return new Vec2(r, phi);
  }
}
```

The method `cartesian` here serves the purpose of preparing the arguments for the constructor call, i.e., to transform the cartesian coordinates to polar coordinates. This method is marked with the `static` keyword. It therefore forgoes the implicit `this` parameter and can be called independent of a specific `Vec2` object. The static method is merely declared in the namespace of `Vec2`. Such a particular static method, that only creates new objects, is also called a **factory method**.

8.6.3 When to Use Immutable Classes?

Which classes should be made immutable and which should not needs to be decided on a case-by-case basis. Since we need to replicate instead of modify objects when we work with immutable classes, this can lead to allocating and initializing a large number of objects. Such operations cost time and memory. A class `Image` that represents a graphical image is for instance not a likely candidate for immutability: Assume the image has 1000×1000 pixels and every pixel requires four bytes. A single image then requires four megabytes of space. If we want to change a pixel in such an image, it is a bad idea to copy $999,999$ pixels to create a *new* image.

The following table summarizes the advantages and disadvantages of immutable vs mutable objects:

	Advantage	Disadvantage
mutable	Efficiency	Mutation through aliases not modular
immutable	“Feels like a value”	Potentially many objects are created

Remark 8.6.2 Many important classes in Java are immutable and use tricky implementations to reduce the inefficiency introduced by copying objects. A prominent example is `String`.

8.7 Inheritance

We have seen how Java supports the programmer for encapsulating classes with visibility modifiers. Our example implementation of `Vec2` now is encapsulated, but it is not yet separated from its specification: If a program declares a variable of type `Vec2`, it invariably refers to our implementation of `Vec2`, most recently using polar coordinates. In larger programs, we do not want our code to refer to a single concrete implementation, but rather an **abstract data type** (**ADT**) that provides a certain **interface**. Code that uses such an abstract data type can then work with *every* implementation of the abstract data type. Several implementations can then be used interchangeably, making the code more modular.

8.7.1 Interfaces

Let us assume that our program uses `Vec2`s with cartesian coordinates as well as `Vec2`s with polar coordinates. Then, any reference to these classes in the program would need to concretely refer to one of the classes:

```java
public class Rectangle {
  private PolarVec2 upperLeft, lowerRight;
  ...
}
```

We however do not want to distinguish between vectors in polar or cartesian coordinates when implementing the above class `Rectangle`. *Any* implementation of the ADT "two-dimensional vector" would do here. To achieve this flexibility, Java provides two important means: **subtypes** and **interfaces**. With an interface, we declare the signature of an abstract data type: All its method signatures with their parameter and return types.

Aside: What we know as "prototype" from C is called "signature" in Java.

```java
public interface Vec2 {
  void setX(double x);
  void setY(double x);
  double getX();
  double getY();
  void scale(double r);
  void add(Vec2 v);
  double length();
}
```

All declared methods of the interface have `public` visibility. Note that interfaces cannot declare member fields. This is by design, because interfaces are intended to not be specific to the internal implementation of an ADT. Concrete implementations can now *implement* this (and an arbitrary number of other) interfaces.

```java
public class PolarVec2 implements Vec2 {
  private double r, phi;

  public PolarVec2(double r, double phi) {
    this.r   = r;
    this.phi = phi;
  }

  public double getX() { return r * Math.cos(phi); }
  ...
  public void scale(double r) {
    setX(r * getX());
    setY(r * getY());
  }
  ...
}

public class CartesianVec2 implements Vec2 {
  private double x, y;

  public CartesianVec2(double x, double y) {
    this.x = x;
    this.y = y;
  }

  public double getX() { return x; }
  ...
  public void scale(double r) {
    setX(r * getX());
    setY(r * getY());
  }
  ...
}
```

PolarVec2 and CartesianVec2 are now subtypes of the type Vec2. We write this as PolarVec2 ≤ Vec2 and CartesianVec2 ≤ Vec2. If A is a subtype of B (we can also say: if A is a subclass of B), references to objects of the class A can also be put into the containers of variables of type B. In our example, this allows us to implement the class Rectangle against the interface Vec2:

```
public class Rectangle {
  private Vec2 upperLeft, lowerRight;

  public Rectangle(Vec2 upperLeft, Vec2 lowerRight) {
    this.upperLeft  = upperLeft;
    this.lowerRight = lowerRight;
  }
  ...
}
```

Aside: ≤ is not a Java operator, we only use A ≤ B as a short hand notation for "A is a subtype of B".

We however cannot create concrete Vec2 objects since the interface Vec2 does not provide a concrete implementation of its methods (as intended). We can use objects of arbitrary *subclasses* of Vec2 to initialize a Rectangle object:

```
public class RectTest {
  public static void main(String[] args) {
    Vec2 v = new PolarVec2(1, Math.PI); // possible since
        Polarvec
                                        //   is a subclass
                                          of Vec2
    Vec2 w = new CartesianVec2(1, 0);   // also possible
    Rectangle r = new Rectangle(v, w);
   ...
  }
}
```

The subtype relation is a partial order: It is

- *reflexive*: For all types $T : T \leq T$
- *transitive*: For all types $U, V, W : U \leq V \wedge V \leq W \implies U \leq W$
- *antisymmetric*: For all types $V, W : V \leq W \wedge W \leq V \implies V = W$

8.7.2 Overriding Methods

If we inspect the classes CartesianVec2 and PolarVec2 closer, we see that they have some code in common. In our reduced example, this is the case for the method scale. In a complete implementation, many more methods would be affected. scale only uses the setters and getters of Vec2 and no more implementation details of PolarVec2 or CartesianVec2. The current implementation duplicates the code of scale. Code duplication is the enemy of every programmer and should be avoided in all cases: If duplicated code is erroneous, it needs to be fixed at multiple source locations that might not be easy to find. Java helps here as well by allowing us to insert a class between the interface `Vec2` and the concrete implementations `CartesianVec2` and `PolarVec2`, implementing parts of `Vec2` but leaving others *abstract*.

```java
public abstract class BaseVec2 implements Vec2 {
  public void scale(double r) {
    setX(r * getX());
    setY(r * getY());
  }
  public double length() {
    double x = getX();
    double y = getY();
    return Math.sqrt(x * x + y * y);
  }
}

public class PolarVec2 extends BaseVec2 {
  ...
}

public class CartesianVec2 extends BaseVec2 {
  ...
}
```

Just as for `Vec2`, we cannot create concrete `BaseVec2` objects since it does not provide implementations for all methods of the interface. Java forces us to mark the *unfinished* state of `BaseVec2` by using the `abstract` keyword in the class declaration. We cannot create objects of abstract classes. `PolarVec2` and `CartesianVec2` then **inherit** from `BaseVec2` and *override* the remaining methods of `Vec2`.

Remark 8.7.1 It is noteworthy that interfaces are only a special kind of class. All vocabulary for classes also applies to interfaces. That Java requires `implements` for interfaces and `extends` for classes is not a conceptional, but rather an aesthetic difference.

When inheriting from a class, we can also **override** methods that already have an implementation in the base class. In this case, the subclass discards the implementation of the overridden method of the base class and uses its own. For example, `PolarVec2` can compute the vector's length more efficiently than it is done in `BaseVec2`:

```java
public PolarVec2 extends BaseVec2 {
  private double r, phi;

  public double length() { return r; }
}
```

These constructs constitute an inheritance hierarchy (Figure 8.7.2). In practice, such hierarchies can easily span several dozen classes. Inheritance hierarchies typically tend to be shallow and broad rather than deep.

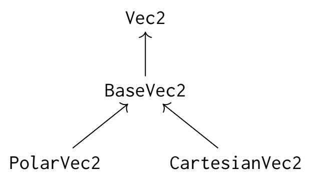


Figure 8.7.2 Inheritance hierarchy of `Vec2`

8.7.3 Calling a Method

The type of a *Variable* in Java (if its type is a class), is only an *upper bound* to the concrete type of the object that it references. The variable declaration `T a;` only states that the container for `a` contains a reference to an object whose type is *at least* `T`. Objects of any subclass of `T` are allowed as well. This property of Java's type system enables the easy replacement of implementations (subclasses). This however means that there can be an ambiguity when calling a method. Consider the above example: The method `length` is implemented by `BaseVec2` and by `PolarVec2`. Which method is actually called depends on the **concrete type** of the object. The concrete type of an object is that whose constructor was used to initialize the object.

```
Vec2 v;
double l;
v = new PolarVec2(1, 0.5);
// calls length from PolarVec2
l = v.length();

v = new CartesianVec2(1, 0);
// calls length from CartesianVec2
l = v.length();
```

The concrete type of the argument `v` therefore decides which method needs to be called. This is another reason for the special syntax of the `this` parameter.

It is important that we understand the difference between the **static type** and the concrete type. The static type is the type that the compiler can determine at compile time of a variable. It is derived directly from the variable's declaration and is, as mentioned previously, an upper bound for the concrete type:

$$\text{concrete type} \leq \text{static type}$$

In the above example, the static type of the variable `v` is `Vec2`, at every method call. The called method is however chosen by the concrete type.

Remark 8.7.3 It is not possible, in general, for the compiler to derive the concrete type of an object at a certain place in the program:

```
if ( /* today is Tuesday */ )
  v = new PolarVec2(1, 0.5);
else
  v = new CartesianVec2(1, 0);
double l = v.length();
```

The run time environment therefore needs to perform additional tasks to keep track of the concrete types of objects when the program is executed.

Remark 8.7.4 With the `instanceof` operator, we can test at run time whether the concrete type of an object is a subclass of a given class. This is a common tool to regain static type information:

```
if (v instanceof PolarVec2) {
  PolarVec2 w = (PolarVec2)v;
}
```

The type conversion `(PolarVec2)v` checks at run time whether the concrete type of the object refered to by the container of `v` is a subtype of `PolarVec2`. If that is not the case, an exception is triggered when the program is executed.

8.7.4 The Class `Object`

Every class in Java inherits from the class `Object`, directly or indirectly. If no base class is provided at a class declaration, Java assumes an implicit `extends Object`.

The class `Object` provides several methods that are important for the interplay with the Java standard library. It might be necessary to override them with suitable custom implementations. We will discuss the more important ones in the following subsections.

8.7.5 `equals`

Java distinguishes objects that are equal from those that are identical:

```
Vec2 v = new PolarVec2(1, 0.5);
Vec2 w = new PolarVec2(1, 0.5);
```

The objects `v` and `w` should be equal, but they are not identical. They are two separate objects that happen to have the same values in their fields. The equality operator == in Java compares only references, therefore

```
System.out.println(v == w);
```

prints `false`.

Aside: We omit the full formulation here: "The objects referenced by the containers...".

Often, the equality of objects is of more interest than their identity. Especially when working with immutable classes, it is common to have several objects that are *equal* at the same time. Java therefore uses equality rather than identity at many places, including the collections framework, which provides implementations of common data structures. This requires a class to define when two of its objects are equal. It needs to override the method `boolean equals(Object)` of the class `Object`:

```
public class BaseVec2 {
  ...
  public boolean equals(Object o) {
    // Stop if o has not at least the type Vec2.
    // This includes the case that o == null.
    if (!(o instanceof Vec2))
      return false;
    // o has at least type Vec2 and is not null
    Vec2 v = (Vec2)o;
    // the vectors are equal if their coordinates are equal:
    return getX() == v.getX() && getY() == v.getY();
  }
}
```

Remark 8.7.5 In the above example, it is essential to use `Object` as the parameter type of `equals`. It is a common mistake to use the current class as the parameter type for the object to compare with. In our example, this would be `boolean equals(BaseVec2)`. Java will not note an error in this case, since this *overloads* the method `equals` rather than *overriding* it (Section 8.8).

Remark 8.7.6 The default implementation of `equals` in `Object` is

```
public boolean equals(Object o) {
  return this == o;
}
```

8.7.6 `hashCode`

The method `hashCode` serves to compute for an object an integer that is as unique as possible. We need this functionality to place an object in a **hash table**. It is only necessary to override `hashCode` if `equals` has also been overridden. The reason for this is that hash tables require that two equal objects always have the same `hashCode`. That is, if, for two objects `p` and `q`, `p.equals(q)` holds, `hashCode` needs to be implemented such that `p.hashCode() == q.hashCode()`.

The Java compiler cannot enforce this condition statically. It is up to the programmer to obey it. The collections framework in Java's standard library uses the `hashCode` method to implement hash maps and sets as discussed in Section 5.3.

8.7.7 `toString`

This method is used to provide a "human-readable" textual representation of an object. Java calls `toString` at several places to obtain this textual representation:

- Many output methods, for instance `PrintStream.println()`, accept a reference of type `Object` and then use `toString` to get a textual representation to print.
- The operator `+` is overloaded with several meanings in Java. If one of its operands has type `String` and the other is at least an `Object`, this operand is converted to a textual representation via `toString`, which is then concatenated to the `String` object.

A reasonable implementation for our example is the following:

```java
public class BaseVec2 {
  public String toString() {
    return "[" + getX() + ",␣" + getY() + "]";
  }
}
```

8.8 Overloading Methods

Java allows us to declare several methods with the same name in a single class. To nevertheless distinguish them properly, methods are not only identified by their name, like in C, but also by their *signature*. Consider the invocation of a method `m`:

$$a.m(e_1, \dots, e_n) \qquad a : U \quad e_i : U_i,\ 1 \le i \le n$$

Which of the overloaded methods `m` could be called here? Remember that the exact method that will be called in the execution can in general only be determined at the programs run time because of overridden methods. Which of a group of overriding methods in a class hierarchy is called is decided based on the *concrete (dynamic) type* of `a` and not its static type.

A first point of consideration is the number of arguments. For the above call, only method declarations whose signature has n parameters (their **arity** is n) can be considered. Next, Java checks whether the static types of the arguments can be converted to the types of the method parameters. Candidate signatures where this is possible are called *matching* signatures. Among those, the Java compiler selects the most specific signature. Since overloaded methods are resolved at compile time, only the static types are relevant.

Definition 8.8.1 More Specific Signature. A signature $(T_1, \dots, T_n)$ is *more specific* than a signature $(U_1, \dots, U_n)$ if $T_i \sqsubseteq U_i$ for all $1 \le i \le n$. Here, $T \sqsubseteq U$ holds if T can be converted to U according to Figure 8.2.2 or if $T \le U$. ◊

Example 8.8.2

1. (`A`, `int`, `int`) is more specific than (`A`, `int`, `float`).

2. (A) is more specific than (B) if $A \leq B$ (as $\leq$ implies $\sqsubseteq$).

□

The method to be called is now selected from the set of methods whose signature is less specific than (or equally specific as) the signature of the call. Java requires that this set of matching signatures contains *one* maximally specific signature, whose method is selected.

Definition 8.8.3 Maximally Specific Signature. Let $\mathcal{S}$ be a set of signatures with an arity of n. $s \in \mathcal{S}$ is a *maximally specific signature* of $\mathcal{S}$ if no $s' \in \mathcal{S} \setminus \{s\}$ is more specific than s. ◊

Definition 8.8.3 does not exclude the existence of more than one maximally specific signature per set of signatures. In this case, the method call would be *ambiguous* and the Java compiler would reject the program with an error message.

Example 8.8.4 Consider the invocation of the method `foo` in the method `call`:

```
class A {
  void foo(int x, int y, float z) {}
  void foo(boolean x, float y) {}
  void foo(int x, float y) {}
  void foo(long x, int y) {}
  void call() { this.foo(2, 2); }
}
```

The argument types are `(int, int)`. `foo` has four declarations with the following signatures:

`(int, int, float)`,   `(boolean, float)`,   `(int, float)`,   `(long, int)`

The first one does not qualify because its arity does not match. Neither does the second one, as the types are not convertible (`int` does not convert to `boolean`). The last two do match, but both are *maximally specific*. Java therefore cannot find a declaration and reports an error. If we added a method

```
void foo(int x, int y) {}
```

its signature would be more specific than `(int, float)` and `(long, int)`. Java then could find a unique maximally specific signature. □

Remark 8.8.5 For the sake of simplicity, we ignored variadic arguments (`...`), boxing/unboxing and type variables here. More details are in Section 15.12.2 of the Java Language Specification [12]. The relation $\sqsubseteq$ is called *Method Invocation Conversion* (see Section 5.3 of the language specification).

8.8.1 Overloading and overriding

If a method `m` is overloaded in a class `A`, we can separately override every overloaded method when we write a class inheriting from `A`. Strictly speaking, we do not override `m`, but `m`$(T_1, \dots, T_n)$: We only override one of the concrete overloaded methods. The compiler selects the overloaded method statically at compile time, while the dynamic method dispatch at run time only decides which overridden variant of the chosen overloaded method is called.

Example 8.8.6 Consider the following classes:

```java
public class A {
  public void m(double X)  {
      System.out.println("A.m(double)"); }
  public void m(boolean x) {
      System.out.println("A.m(boolean)"); }
}

public class B extends A {
  public void m(int x)     {
      System.out.println("B.m(int)"); }
  public void m(boolean x) {
      System.out.println("B.m(boolean)"); }
}
```

To understand which method overrides which, consider the following table:

Table 8.8.7 Overloaded and overridden methods

Class	Methods		
A	`A.m(double)`	`A.m(boolean)`	
B		`B.m(boolean)`	`B.m(int)`

Methods in a row with the same name overload each other. A method in a column overrides the method in the same column of a row above. Hence, here only `B.m(boolean)` overrides the method `A.m(boolean)`. Consider the following program fragments that call `m` in different constellations:

1.
   ```java
   A a = new A();
   a.m(5);
   ```
   prints `A.m(double)`. Trivial, since the concrete type is equal to the static type.

2.
   ```java
   A a = new B();
   a.m(5);
   ```
   prints `A.m(double)`. The static type of `a` is `A`, but the concrete type is here `B`. However, `B` does not override the method `A.m(double)`, but only overloads the methods `m(double)` and `m(boolean)` again with the parameter type `int`. Since the overloaded method is selected by the static type (here: `A.m(double)`), the method `m(int)` in the concrete type `B` is irrelevant.

3.
   ```java
   B b = new B();
   b.m(5);
   ```
   prints `B.m(int)`. The static type is now `B` and `m` is overloaded three times in `B`.

□

8.9 The Java Collections Framework

One very helpful component of the Java standard library is the extensive Java Collections Framework[17] that provides implementations of many commonly used data structures. Some of them we have already discussed in Chapter 5. In this section, we only give a very high-level overview and leave out many fine-grained details that the avid reader can obtain from the official documentation.

[17] `docs.oracle.com`

The collections framework is organized as a class hierarchy that provides interfaces for many common abstract data types and several implementations of them. The top interface is called `Collection<T>`[18] which essentially stands for a multi-set, i.e. something similar to a set except for the fact that elements can appear multiple times in it. It is different from a list in that elements are not ordered. A collection provides, among other things, the following operations: add, remove, (check if it) contains (an element), size, iterate (over all elements):

```
interface Collection<E> {
  boolean add(E elem);      // Ensure that the collection
                            // contains element after call
  boolean remove(E elem);   // Remove a single instance of
                            // element if coll contains it
  void clear();             // Remove all elements
  int size();               // Get size
  Iterator<E> iterator(); // Get an iterator
  // more methods ...
}
```

From the base interface collection, more detailed interfaces and classes are derived: lists, sets, queues, deques (double-ended queues). Queues are data structures where elements can be added to and removed from. Depending on the kind of queue this happens in a first-in last-out style similar to a list or according to a completely different policy as in priority queues. We will not go into further detail about queues here. In this text, we focus on lists and sets because we have already discussed some of them. We have discussed array lists in Subsection 5.1.1 and doubly-linked lists in Subsection 5.1.3. The Java implementation follows our discussion quite closely. The tree sets are self-balancing binary search trees and hash sets are based on the techniques discussed in Section 5.3.

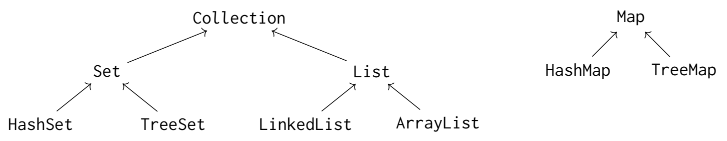


Figure 8.9.1 Coarse overview over some of the most important classes in the Java Collections Framework. We left out some intermediary classes and other data structures we have not discussed here.

Lists[19] extend collections by imposing an order on the inserted elements. Adding will append to the list, removing will remove the first instance of the list. Additionally, one can also obtain the i-th element of a list.

```
interface List<E> implements Collection<E> {
  E get(int index);
  // more methods ...
}
```

Sets[20] don't add any new relevant methods but enforce "set semantics" most importantly that adding an element that is already contained a second time does not change the set. Furthermore, there is also no `get(int index);` method because sets are not ordered.[21]

[18] docs.oracle.com
[19] docs.oracle.com
[20] docs.oracle.com
[21] There is however an interface `OrderedSet`.

8.9.1 Maps

A map assigns elements from one set, so-called *keys*, an element from another set, called the *value* set. Similar to mathematical maps, each key can be assigned to at most one value. Sets can be seen as a special case of maps where the value set is a singleton set (containing only one element).

In the Java collection framework, maps are no collections but form a class hierarchy of their own with the interface `Map<K, V>` at its top.

```java
interface Map<K, V> {
  V put(K key, V value);  // Associate key with val
                          // Return old association or null
  boolean remove(K key);  // Remove association from key
  int size();             // Number of keys in map
  V get(K key);           // Get value for key or null
  boolean containsKey(K key); // check if there's an assoc
      for key
}
```

8.9.2 Hash Sets

We discuss data structures and algorithms for hashing in Section 5.3. Here, we discuss specific details that are relevant to use the hash sets and maps in the Java collection framework.

When adding an object `o` to a hash table (either a set or a map), Java invokes `o.hashCode()` (which is defined in `Object`) to compute the hash code for the object. Since Java collections also use `equals` when searching for an entry in a collection, `equals` and `hashCode` have to be consistent in the following way:

Definition 8.9.2 hashCode Consistency. Assume that `q` and `p` are both objects of a class T. Then, whenever `p.equals(q) == true` it must hold that `p.hashCode() == q.hashCode()`. ◊

Note that the opposite is in general not always true since it would imply that T is isomorphic to `int`.

As a rule of thumb, you should, whenever you want to override `equals` or `hashCode` also override the other and check that both are consistent with respect to Definition 8.9.2.

To understand why it is important to maintain hashCode consistency, consider this class which violates hashCode consistency:

```java
public class Fraction {
  private int numerator, denominator;

  public boolean equals(Object o) {
    if (! (o instanceof Fraction))
      return false;
    Fraction f = (Fraction) o;
    return this.numerator * f.denominator
        == f.numerator * this.denominator;
  }

  public int hashCode() { return this.numerator; }
}
```

The two fractions 1/2 and 2/4 are clearly equal according to the implementation of `equal` in `Fraction`. However, in each hash table longer than 2, they end up in different buckets which has the effect that after inserting 1/2, `contains(new Fraction(2, 4))` will return false.

8.9.3 Iteration

One aspect that is particularly noteworthy about collections is iteration. Each collection can create **iterators** which are objects that capture the state of an iteration over all elements in a collection. Based on the particular collection (e.g. list, set, etc.) the order of iteration is defined or not. An iterator[22] provides three main methods:

```java
interface Iterator<E> {
  boolean hasNext(); // check, if the iterator is valid
  E next();          // if valid, return current element
                     // and proceed to next
  void remove();     // if valid, remove current element
                     // and proceed to next
}
```

For example, the iterator of an array list could look like this:

```java
public class ArrayList<E> implements List<E> {
  private E[] elements;
  private int size;

  // ...

  public Iterator<E> iterator() {
    return new Iterator() {
      private int i = 0;
      public E next() {
        return elements[i++];
      }

      boolean hasNext() {
        return i < size;
      }

      // ...
    };
  }
}
```

Since iteration is very frequent, Java offers a special for-loop to iterate over collections (and also arrays):

```java
Set<Vec2> s = ...;

for (Vec2 v : s) {
  System.out.println(v.length());
}
```

This is (more or less) equivalent to this more verbose piece of code:

```java
Set<Vec2> s = ...;

for (Iterator<Vec2> it = s.iterator(); it.hasNext(); ) {
  Vec2 v = it.next();
  System.out.println(v.length());
}
```

[22] docs.oracle.com

8.9.4 Genericity

The collections framework extensively uses genericity in the form of type variables. These provide an upper bound on the type of the objects that can be added to the collection. For example:

```java
Set<Vec2> s = new HashSet<Vec2>();
```

The type `Set<Vec2>` makes sure that only objects that are a `Vec2` can be added to `s`. Accordingly, when e.g. iterating over the set, the type variable ensures that the type of each object in the set is at least `Vec2`.

Because type variables have only been added to Java later, they have not made it into Java byte code. The Java compiler replaces type variables by `Object` when compiling the Java program into Java byte code in a process called **type erasure**. This is the cause of some inconsistencies that occasionally cause trouble. Most prominently has Java been designed with *covariant* arrays, i.e. you can write `Vec2[] o = new PolarVec2[10];`. This requires the language to perform dynamic type checks when you assign to an array. For example, when writing `o = new CartesianVec2(...);` an `ArrayStoreException` has to be thrown. This in turn requires that the array knows its element type at run time to be able to perform this check. However, since type variables are erased to `Object`, you cannot create arrays with type variables, i.e. the following is not possible: `T[] a = new T[10];`

8.10 Using Object Orientation Properly

Object-oriented programming languages help to solve an important problem for developing large software systems: easy extensibility. “Easy” here means that extending the system only requires changes at a small number of confined program locations. This is important since modifications that address many program locations are very prone to mistakes: A developer that performs such modifications needs to understand the entire code affected by the changes to ensure that the modification is correct and complete. Object-oriented languages facilitate this by providing means that, if used properly, confine the required changes for an extension of the program. The following properties of object-oriented languages are central for this goal:

- The *separation of interface and implementation* is helpful for implementing abstract data types. This supports extensibility since code can refer to the interface rather than a concrete implementation. Subtyping ensures that all extensions provide the functionality required by the interface.

- The *concentration of data and corresponding code in a class* prevents implementation details from being scattered among large parts of the code and from being used by code parts that actually do not need them.

We want to discuss these aspects now by means of a simple example that we implement in functional (OCaml), imperative (C), and an object-oriented (Java) language. The goal of looking at three different languages from three different “paradigms” is to contrast the support each language offers for implementing our small example and to compare and relate object-oriented programming to other programming paradigms.

Our program shall represent the abstract syntax trees of a simple arithmetic expression language. This language consists only of variables and the addition of expressions:

Category		Abstract Syntax	Concrete Syntax	Description	
$Exp \ni e$	$::=$	$\mathtt{Add}[e_1, e_2]$	$e_1 + e_2$	Addition	(8.1)
	$\mid$	$\mathtt{Var}_x$	x	Variable	

This example is representative for many situations in programming: There is a category of "things" and different *variants* of them. Here the category is the syntactical category of expressions and the variants are addition and occurrence of a variable. But you can also think of the category "element of a graphical user interface" and the variants would be "button", "checkbox field", "text entry field" and so on. Or, just to give another example, the category could be "input/output stream" and the variants could be "file", "inter-process pipe", "network socket", and so on.

Coming back to our example, we want to provide two operations for these expressions: Creating a textual representation of an expression as a string and evaluating an expression. Listing 8.10.1, Listing 8.10.2, and Listing 8.10.3 show the implementation of this data structure and the operations in OCaml, C, and Java.

OCaml. Let us first consider the implementation in OCaml. Functional languages traditionally have good support for implementing data types like the one we need for the AST of our expression language. They provide **Algebraic Data Types** (not to be confused with *abstract* data types) which allow the programmer to specify the variants of which the data type is composed. Here `Add` and `Var`. The operations that operate on values of that data type typically use `match` constructs to discriminate the individual cases.

```
type exp =
  | Var of string
  | Add of exp * exp

let rec to_string e =
  match e with
  | Var v -> v
  | Add (l, r) -> (to_string l) ^ "␣+␣" ^ (to_string r)

let rec eval e st =
  match e with
  | Var v -> st v
  | Add (l, r) -> (eval l st) + (eval r st)
```

Open in Browser

Listing 8.10.1 Implementation of the abstract syntax of the simple expression language in OCaml.

C. Let us now consider the implementation in C (Listing 8.10.2). The C implementation exposes the low-level details of algebraic data types and shows how they might be implemented in a functional language "under the hood". The struct `exp_t` contains a *union* containing the data for the two variants of our ADT. The member field `op` determines whether the expression is an addition or the occurrence of a variable. Which variant is represented is set by the corresponding constructor (`exp_init_var` and `exp_init_add`).

```c
struct env_t;
typedef struct env_t env_t;
int env_get(env_t const *e, char const* name);

typedef enum {
    ADD, VAR,
} operator_t;

typedef struct exp_t {
    operator_t op;
    union {
        struct exp_t *opnds[2];
        char const* var_name;
    };
} exp_t;

exp_t* exp_init_add(exp_t* t,
                    exp_t* l, exp_t* r) {
    t->op = ADD;
    t->opnds[0] = l;
    t->opnds[1] = r;
    return t;
}

exp_t* exp_init_var(exp_t* t, char const* n) {
    t->op = VAR;
    t->var_name = n;
    return t;
}

int exp_eval(exp_t const* t, env_t const* e) {
    switch(t->op) {
    case ADD:  return exp_eval(t->opnds[0], e)
                    + exp_eval(t->opnds[1], e);
    case VAR:  return env_get(e, t->var_name);
    }
}

void exp_print(exp_t const* t, FILE* f) {
    switch(t->op) {
    case ADD:  exp_print(t->opnds[0], f);
               fprintf(f, "␣+␣");
               exp_print(t->opnds[1], f);
               break;
    case VAR:  fputs(t->var_name, f);
               break;
    }
}
```

Open in Browser

Listing 8.10.2 Implementation of the abstract syntax of the simple expression language in C.

The implementations for expression evaluation (in `exp_eval`) and printing (in `exp_print`) use this field to execute the corresponding variant of the code. These two functions thus contain the corresponding code for all expression variants. A variant's implementation is therefore scattered over many places in the C code: The *enum* that lists all variants, a member field in the *union* for the data and a branch in every function implementing an operation of the data type.

Since C does not have inheritance, the `exp_t` struct collects fields for all subclasses. The lack of classes requires branches in case distinctions for the method implementations (see `exp_eval` and `exp_print`). These functions can access all fields, meaning that there is no encapsulation between the `Add` and `Const` classes.

Java.

```java
public interface Env {
    int get(String varName);
}

public interface Exp {
    public int eval(Env e);
    public String toString();
}

public class Add implements Exp {
    private Exp left, right;

    public Add(Exp left, Exp right) {
        this.left = left;
        this.right = right;
    }

    public int eval(Env e) {
        return left.eval(e) + right.eval(e);
    }

    public String toString() {
        return "" + left + "␣+␣" + right;
    }
}

public class Var implements Exp {
    private String name;

    public Var(String name) {
        this.name = name;
    }

    public int eval(Env e) {
        return e.get(this.name);
    }

    public toString() {
        return this.name;
    }
}
```

Listing 8.10.3 Implementation of the abstract syntax of the simple expression language in Java.

The Java implementation uses the object-oriented way of implementing algebraic data types. An interface declares all operations that can be applied to expressions. Every class implementing this interface corresponds to a variant. Every aspect of this variant (implementation of operations, initialization, data) are gathered in the class. Subtyping ensures that such a class can be used whenever an object of type `Exp` is expected. The value of the field op in the C implementation corresponds to the concrete type of the object in the Java implementation. The resolution of the overridden methods replaces the case distinctions in the C functions `exp_eval` and `exp_print`. The code that handles this internally is, without further ado by the programmer, inserted automatically by the compiler.

8.11 The Expression Problem and the Visitor Pattern

In this section, we will investigate what is known as the "expression problem" [14]. The expression problem refers to a fundamental and old problem in software engineering that is concerned with extending a data type (like the one we used in the previous section 8.10) in *two* dimensions: (1) adding new variants to a data type and (2) adding new functionality to all variants. It is interesting to study the expression problem because it exposes specific differences in functional and object-oriented programming languages which both tend to support one of the two dimensions of extensions better than the other.

Let us stick to the example of modeling the AST of a small expression language (8.1) and let us reconsider the OCaml (Listing 8.10.1) and Java (Listing 8.10.3) implementations.

8.11.1 Extensions

Now, let us discuss two extensions to this data type. The first extension is to add a new variant, let's say for representing constants in the expression. The second extension will be to add a new operation to all variants, let's say formatting the AST as an XML document. Both languages handle these extensions differently.

Adding a new Variant. Let us add a constant to our simple language. In the Java implementation this corresponds to a new class `Const` that implements the `Exp` interface. In this class, all relevant operations are implemented as methods. The existing code does not need to be modified, hence this extension is local.

```java
class public Const implements Exp {
    private int value;

    public Const(int value) {
        this.value = value;
    }

    public int eval(Env e) {
        return this.value;
    }

    public String toString() {
        return "" + this.value;
    }
}
```

In the OCaml implementation however, the extension is not local because it involves changing the existing code at various places. First, a new variant has to be added to the `exp` type. Then, the `match` branches of each existing function that operates on `exp` values have to be extended:

```ocaml
type exp =
  | Var of string
  | Add of exp * exp
  | Const of int

let rec to_string e =
  match e with
  | Var v -> v
  | Add (l, r) -> (to_string l) ^ "␣+␣" ^ (to_string r)
  | Const i -> string_of_int i

let rec eval e st =
  match e with
  | Var v -> st v
  | Add (l, r) -> (eval l st) + (eval r st)
  | Const i -> i
```

Open in Browser

Adding a new Operation. When adding a new operation, the locality of change is just the other way around. Let us explore this by extending our implementation by a feature that allows for formatting an AST as an XML document. In the OCaml implementation, it is sufficient to add a new function `to_xml` that contains the implementation of this feature for all existing variants. This is a *local* change because none of the existing code needs to be modified.

```
 | Const i -> i

let rec to_xml e =
  match e with
  | Var v -> "<var>" ^ v ^ "</var>"
  | Add (l, r) -> "<add>" ^ (to_xml l) ^ (to_xml r) ^
      "</add>"
  | Const i -> "<const>" ^ (string_of_int i) ^ "</const>"
```

Open in Browser

In the Java implementation however, this extension is *non-local*. It requires to add a new method `toXML` to the interface and each class that implements it.

```
public interface Exp {
    public int eval(Env e);
    public String toString();
    public String toXML();
}

public class Add implements Exp {
    private Exp left, right;
    // ...
    public String toXML() {
      return "<add>" + left.toXML() + right.toXML() +
          "</add>";
    }
}

public class Var implements Exp {
    private String name;
    // ...
    public String toXML() {
      return "<var>" + name + "</var>";
    }
}
```

```java

public class Const implements Exp {
    private int value;
    // ...
    public String toXML() {
      return "<const>" + value + "</const>";
    }
}
```

The expression problem now is a challenge in programming language design to support both kinds of extensions in a local way.

8.11.2 The Visitor Pattern

One way of making the second extension (adding new operations) in an object-oriented language more local and thus more bearable is the **visitor pattern**[23]. The visitor pattern factors out the extension of a class hierarchy to a single extension point, the so-called *visitor*.

Let us make a first attempt to extend the `Exp` data type with a new operation in a *local way* by reusing the example above: adding an XML printer.

```java
public class XMLFormatter {
    public String format(Exp e) {
        if (e instanceof Const) {
            Const c = (Const) e;
            return "<const>" + c.getValue() + "</const>";
        }
        else if (e instanceof Var) {
            Var v = (Var) e;
            return "<var>" + v.getName() + "</var>");
        }
        else if // etc.
    }
}
```

This code isolates the extension in one class `XMLFormatter`. However, the if-then-else case distinction using `instanceof` seems very awkward because it involves dynamic casts which should only be a last resort. This code is also bad for several other reasons. For one and maybe most problematic: it is easy to forget cases without causing any warning by the compiler. Furthermore, the code is inefficient, since potentially many `instanceof` conditions need to be checked before the matching case is found. Lastly, the order of the `if`s affects the run time in ways that can lead to surprising performance behaviors.

The better solution is to cleanly separate the code for each subclass into its own method:

```java
public class XMLFormatter {
    public String format(Const c) {
        write("<const>" + c.getValue() + "</const>");
    }
    public String format(Var v) {
        write("<var>" + v.getName() + "</var>");
    }
    // the remaining cases...
}
```

[23]The visitor pattern is called a *pattern* because it appears in a larger library of so-called design patterns that are best practices in the design of (object-oriented) software to solve specific, recurring problems.

We now however need to ensure that these methods are also called. This is problematic since we might only know the static type `Exp` and not the concrete type at the desired call site:

```java
public class Main {
    public static void main(String[] args) {
        Exp e = constructExpressionFromInput();
        System.out.println(new XMLFormatter().format(e));
    }
}
```

This code will not work since no `format(Exp)` method is available in `XMLFormatter`. We would therefore again need to introduce a case distinction to obtain the concrete type of the objects, insert an explicit type cast, and then call the corresponding method which would bring us back to square one.

We can however eliminate the case distinction with an indirection. The object `e` from the previous code section does know its concrete type. We therefore extend the interface `Exp` with a single additional method `accept`, which only calls another method `visit` of an interface `ExpVisitor` with `this` as an argument.

```java
public interface ExpVisitor<T> {
    T visit(Const c);
    T visit(Var c);
    T visit(Add c);
}

public interface Exp {
    <T> T accept(ExprVisitor<T> v);
}

public class Const implements Exp {
    // ...
    <T> public T accept(ExpVisitor<T> v) {
        return v.visit(this);
    }
}

public class Var implements Exp {
    // ...
    <T> public T accept(ExpVisitor<T> v) {
        return v.visit(this);
    }
}

public class Add implements Exp {
    // ...
    <T> public T accept(ExpVisitor<T> v) {
        return v.visit(this);
    }
}
```

`XMLFormatter` now implements the `ExpVisitor` interface and provides methods for every case:

```java
public class XMLFormatter implements ExpVisitor<String> {
    // ...
    public String visit(Const c) {
        return "<const>" + c.getValue() + "</const>";
    }

    public String visit(Var v) {
        return "<var>" + v.getName() + "</var>";
    }

    public String visit(Add b) {
        return "<add>"
             + b.getLeft().accept(this)
             + b.getRight().accept(this)
             + "</add>";
    }
}
```

The main method can then pass an `XMLFormatter` object as an argument to the `accept` method:

```java
public class Main {
    public static void main(String[] args) {
        Exp e = constructExpressionFromInput();
        e.accept(new XMLFormatter());
    }
}
```

The whole point of the visitor is to use dynamic type information to select the appropriate `accept` method in the visitor object. At first sight, it seems superfluous to implement the `accept` method in every subclass of `Exp`. However, this is where the selection is made: The static type of `this` in each method of a class T is T. In the `accept` methods, the static type of `this` is used to select one of the overloaded `visit` methods. Hence, if the *dynamic* type of an object o is T in some execution, `o.accept(visitor)` will call the method `visit(T obj)` of the visitor class. This technique avoids an `instanceof` if-then-else cascade by using the accept/visit method call sequence to promote dynamic type information into static one.

To summarize, object-oriented languages have an inherent weak spot when extending a class hierarchy with new operations. They force the programmer to perform non-local changes to the code because the new operation has to be added to each subclass. The visitor pattern is suitable to solve this problem because it allows to factor out the new functionality into a separate *visitor* class. It uses the accept/visit call sequence to dispatch the desired `visit` method in the visitor object based on the *dynamic* type of the "visited" object. Thereby it nicely implements a match-like operation over the dynamic type of an object without resorting to if-then-else cascades.

8.12 Object-Oriented Programming with C

In the previous section, we have seen the differences between an object-oriented language and a purely imperative language like C. In this section, we outline how object-orientation can be realized in C.

Consider the example in Listing 8.10.3 and Listing 8.10.2. For simplicity's sake, we assume that a class can only inherit from/implement a single class/interface. Allowing to implement several interfaces would slightly complicate the implementation without contributing fundamentally to the understanding.

Instead of writing functions where every class is handled as a branch (like in Listing 8.10.2), we create a function for every method. For every class, there is a table of functions (more specifically: function pointers). This table is called **virtual method table**, which is often abbreviated as VMT, vtable, or vtab. The VMT contains for each method of the class (and every super class) the address of the corresponding function. Since every class has its own VMT, the address of the VMT identifies the class uniquely. It corresponds to the "tag" `op` of the struct `exp_t` in Listing 8.10.2. Every object obtains during its initialization (where the concrete type is known) a pointer to the VMT of the class in a field that is hidden to the programmer (`vtab` in Listing 8.12.1). Since we need to access the VMT without knowing the concrete type, the pointer to the VMT is at the same position for *every* object (usually at offset 0). This allows us to identify the concrete type of any object at run time.

The address of the virtual method required at a given call site results from a two-step process: First, the right entry in the VMT needs to be determined. This information is given by the method name (and the signature, in case of method overloading) and can therefore be derived *statically* at compile time. In our example in Listing 8.12.1, the selected entry corresponds to a field in the `...vtab_t` structs. Java's static semantics guarantees that there is an entry for the called method in *every* possible VMT. In the second step, the concrete VMT is determined. Since the VMT address has been written to the object's hidden `vtab` field during initialization, we only need to load this address *at run time* to obtain the right VMT.

One such call occurs in `exp_add_eval` in Listing 8.12.1 in the `return` statement. The expression `l->vtab->eval` determines the address of the overridden method `eval` via the VMT.

```c
typedef struct {
    int  (*eval)(void* this, env_t const* e);
    void (*print)(void* this, FILE* f);
} exp_vtab_t;
typedef struct { exp_vtab_t* vtab; } exp_t;

// Var and Add have the same VMT, as they add no methods.
typedef exp_vtab_t exp_add_vtab_t;
typedef struct {
    exp_add_vtab_t const* vtab; exp_t* opnds[2]; }
        exp_add_t;

// Const has one more method. The methods need to be listed
// in the same order as in exp_vtab_t.
typedef struct {
    int  (*eval)(void* this, env_t const* e);
    void (*print)(void* this, FILE* f);
    int  (*get_value)(void* this);
} exp_const_vtab_t;
typedef struct {
  exp_const_vtab_t const* vtab; int value; } exp_const_t;

int exp_add_eval(void* this, env_t const *e) {
    exp_add_t* add = this;
    exp_t* l = add->opnds[0]; exp_t* r = add->opnds[1];
    return l->vtab->eval(l, e) + r->vtab->eval(r, e);
}

// VMT for Add
static const exp_add_vtab_t exp_add_vtab = {
    exp_add_eval, exp_add_print };

// Constructor for Add
exp_add_t* exp_init_add(exp_add_t* this, exp_t* l, exp_t*
    r) {
    this->vtab = &exp_add_vtab;
    this->opnds[0] = l; this->opnds[1] = r;
    return this;
}

int exp_const_eval(void* this, env_t const *e) {
    exp_const_t* t = this;
    return t->value;
}

// VMT for Const
static const exp_const_vtab_t exp_const_vtab = {
    exp_const_eval, exp_const_print, exp_const_get_value };

// Constructor for Const
exp_const_t* exp_init_const(exp_const_t* this, int value) {
    this->vtab  = &exp_const_vtab;
    this->value = value;
    return this;
}
```

Open in Browser

Listing 8.12.1 “Object-oriented” variant in C for the example from Listing 8.10.3.

Chapter 9

A Simple Compiler

In this chapter, we will develop a very simple **compiler** from C0 to MIPS. The goal is to give some introduction into how compilers work on a very high-level and improve our understanding of how programming languages are implemented on machine code. As a side effect, we'll also be able to put some of the object-oriented modelling skills we have discussed in Section 8.10 to good use.

A compiler translates a program, given as a stream of characters, into a machine code program that is semantically equivalent to the source program. We have gained some impression on how to formally define the semantics of a programming language in Chapter 6. Here, we will not discuss the preservation of semantics formally because it is out of the scope of this text. It would require defining a formal semantics of machine code as well and then dwell into formal notions of program equivalence which is the subject of more advanced texts. Figure 9.0.1 shows the architecture of a simple compiler.

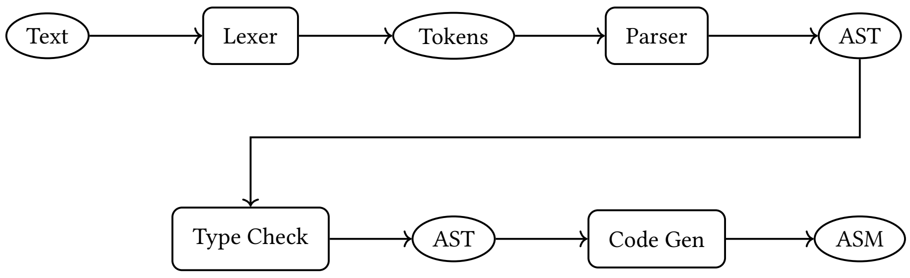


Figure 9.0.1 Schematic architecture of a very simple compiler. The ellipses indicate data. The boxes are passes of the compiler. Modern compilers typically use several other program representations in addition to the AST. They also "optimize" the program (= rewrite it in a semantics-preserving way) with the goal of improving its run time.

Lexing and parsing (which we discuss in the next section) are part of **syntactic analysis**, the so-called **front-end** of the compiler. Its goal is to check if the input character stream adheres to the concrete syntax of the programming language and, if that is the case, construct the abstract syntax tree. **Type checking** is a static program analysis that determines if the program is **well-typed** and elaborates the types for the program. Well-typedness is an important part of the **static semantics** of the programming language.[24] Code generation is part of the compiler's **back-end** and generates code for the program in some machine language. The back-ends of modern compilers can be very complicated because they perform sophisticated program

[24]The static semantics can comprise additional things, for example, making sure that every local variable is assigned before read.

analyses and transformations to optimize the performance (or size) of the machine code program. Here, we concentrate on a very simple code generation technique, called **syntax-directed code generation** that produces less efficient code but is very easy to understand and implement.

It may also be worthwhile mentioning that a substantial part of modern compilers is the optimizer in the “middle-end” which performs various program analyses and transformations in order to make the program more efficient. These optimizations are mostly done independent of the source and target language such that they can be reused when compiling from and to different languages.

9.1 A Brief Overview of Syntactic Analysis

The purpose of syntax analysis is to check if the stream of input characters that constitute the input program adheres to the *concrete syntax*[25] of the programming language and, if that is the case, to construct the *abstract syntax tree*. The abstract syntax tree is the first data structure of a compiler that represents the program to be compiled. It is a so-called **program representation**. Because this text focuses on two compiler phases that operate on the abstract syntax tree, we do not cover syntax analysis in detail here but only give a very high-level overview.

In principle, syntax analysis consists of two steps: lexing and parsing. Lexing forms “words” out of the input characters and parsing checks if the stream of “words” is syntactically correct.

Lexing. The first step in syntax analysis is **lexing**. The name lexer comes from the greek word lexis which means “word”. The job of the lexer is to form “words” from the sequence of input characters. These words are called **tokens**. Hence, the **lexer** translates the stream of input characters that constitute the program text into a stream of **tokens** which represent the “words” of the program. Each token consists of

1. a lexical category (“identifier”, “addition sign”, “open parentheses”, etc.).
2. the original text (if not apparent from the category)
3. the source code coordinates (file, line, column) for error reports.

All other things (mostly whitespace and comments) are discarded by the lexer because they are not relevant for the syntactic correctness of the program. Figure 9.1.1 shows an example program with its corresponding token stream.

```
q = 0;               ID("q") ASSIGN INT_CONST("0") SEMI
r = x;               ID("r") ASSIGN ID("x") SEMI
while (y <= r) {     WHILE LPAREN VAR("y") LE VAR("r") RPAREN LBRACE
    r = r - y;       ID("r") ASSIGN ID("r") MINUS ID("y") SEMI
    q = q + 1;       ID("q") ASSIGN ID("q") PLUS INT_CONST("1") SEMI
}                    RBRACE
```

Open in Browser

Figure 9.1.1 An example program and its corresponding token stream. Each token consists of a name and, if important, the original text in the program that corresponds to the token.

The lexer can already detect some simple errors: If a sequence of characters cannot constitute a valid token of the programming language, the lexer can report

[25] See also Section 6.1 for a somewhat more detailed discussion of the syntax of programming languages.

this error and abort the compilation process. For example `123abc` is not a valid token in C or Java because numbers cannot contain letters and identifiers must start with a non-numeral character. Lexers can however not detect structural errors such as `123 abc + - *`. Each part of that text is a valid C token, however the way they are combined is not correct. It is the job of the parser to identify such errors.

Parsing. The parser analyzes the token stream based on the definition of the concrete syntax and creates, if the program is syntactically correct, the abstract syntax tree. If the token stream does not adhere to the concrete syntax of the programming language, the parser reports (possibly multiple) error messages that point to the syntax error(s). In this text, we don't discuss concrete syntax analysis techniques and refer to standard compiler text books such as [16].

Implementing the AST. The abstract syntax of C0 has two main categories: statements and expressions. As already discussed in Section 8.10, we create an interface per syntactic category (statement and expression) and subclasses for each syntactical element (while loop, if-then-else, add operator, occurrence of a variable, etc.). Our small compiler will consist of a type checker (Section 9.2) and a code generator (Section 9.3) which we will discuss in the following sections. Both operate on the abstract syntax tree and will therefore be implemented as methods to the respective AST classes.

```java
public interface Statement {
  void checkType(...);
  void genCode(...);
}

public interface Expression {
  Type checkType(...);
  void genCode(...);
}
```

9.2 Type Checking

We have discussed the static semantics of C0 in Section 6.6. The static semantics for expressions is modelled as a relation that relates a type environment with expressions and a type. The static semantics for statements relates the type environment with statements. The static expression semantics is *deterministic* (or functional) which means that each type environment and each expression are related to exactly one type. This means that we can easily implement it as a method (in the interface `Expression`) that takes a type environment (here implemented by a class `Scope` which we will discuss below), an expression and delivers a type. Statements can be handled similarly: we implement the static statement semantics as a method (in the interface `Statement`) that takes a type environment as a parameter and check if a statement is well-typed. Both methods generate an error (using a `Diagnostic` object) that is shown to the user if the statement/expression is not well-typed.

```java
public interface Expression {
  Type checkType(Diagnostic d, Scope d);
}
```

The following code shows how type checking looks like for a simple add expression. It effectively implements rule [TArith] in Section 6.6. The code determines the types of the sub-expressions and computes the type of the expression based on this information.

```
public class Arith implements Expression, Locatable {
  // ...
  public Type checkType(Diagnostic d, Scope s) {
    Type l = getLeft().checkType(d, s);
    Type r = getRight().checkType(d, s);
    if (!l.isIntType())
      d.printError(this, "...");
    if (!r.isIntType())
      d.printError(this, "...");
    return Types.getIntType();
  }
}
```

Scope. The type environment is implemented by the class Scope that maps an identifier to the AST node of its declaration in the *current* scope. Scope objects are created during the type checking process: Whenever we enter a new block, we create a new scope to collect the variable declarations in that scope (see rule [TBlock] in Section 6.6). To access the declarations of variables in outer scopes, a scope also links to its parent scope via a reference, making it effectively a stack of scopes. When type checking for this block is finished, we can pop the scope of the block from the scope stack. This corresponds naturally to the scope nesting of the programming language. The following code shows the handling of scopes when type checking a block:

```
public class Block implements Statement, Locatable {
  private final <Statement> body;

  public void checkType(Diagnostic d, Scope parent) {
    Scope scope = parent.newNestedScope();
    // local variables are added by declaration statements
    // in the statement list that constitutes the block's
       body
    for (Statement s : body)
      s.checkType(d, scope);
  }
}
```

When looking up the type of an identifier we have to consult the top scope if it has a declaration for that identifier. If so, we return it, if not, we look in the parent scope and repeat the process until we reach the root scope. If we do not find a definition for the identifier there, we have to signal an error because it means that an identifier is used without being declared.

```
{ // Scope 1
  int x;
  int y;
  x = 1;
  y = 1;
  { // Scope 2
    int z;
    z = x + y;
  }
  { // Scope 3
    int y;
    y = x + 1;
  }
}
```

Open in Browser

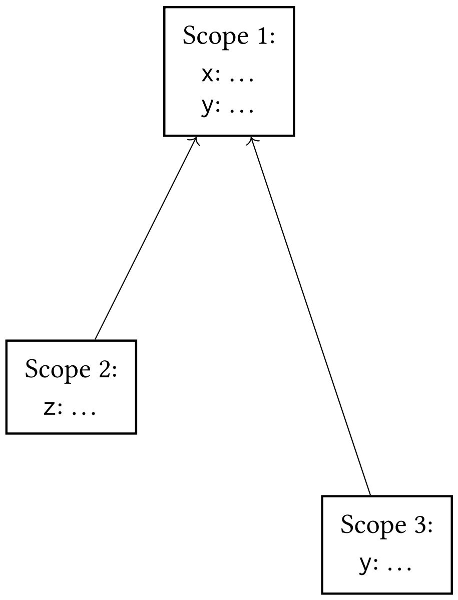


Figure 9.2.1 A C0 with multiple nested blocks. For each block there is a type environment / scope. The nesting of the blocks corresponds to a tree of scopes. The type environment of a block only lists the innermost variable declarations and points to the next outer scope via a pointer. Note that each scope maps the declared identifier to the AST node of the declaration (which is not shown in the figure).

The following code shows a sample implementation for the class Scope.

```
public class Scope {
  private final Map<String, Declaration> table;
  private final Scope parent;

  public Scope() {
    this(null);
  }

  private Scope(Scope parent) {
    this.parent = parent;
    this.table  = new HashMap<String, Declaration>();
  }

  public Scope newNestedScope() {
    return new Scope(this);
  }

  public void add(String id, Declaration d)
    throws IdAlreadyDeclared {
    if (table.contains(id))
      throw new IdAlreadyDeclared(id);
    table.put(id, d);
  }

  public Declaration lookup(String id) throws IdUndeclared {
    // ...
  }
}
```

When visiting an AST node during type checking, it is advisable to store references to the significant declaration in the AST node that represents the using occurrence of an identifier. By this reference, the type can later on be looked up again. This may be important for successive passes like code generation which we discuss in the next section. In general, the AST node of the declaration stands for the variable itself and the compiler may want to associate other information with variables in later stages. The following code shows an example implementation of the AST node that represents the using occurrence of a variable.

```java
public class Var implements Expression, Locatable {
  private String id;
  private Declaration decl = null;

  public Type checkType(Diagnostic d, Scope s) {
    try {
      this.decl = s.lookup(id);
      return this.decl.getType();
    }
    catch (IdUndeclared e) {
      d.printError(this, "Variable␣" + id + "␣not␣
          declared");
      return Types.getErrorType();
    }
  }
}
```

9.3 Syntax-Directed Code Generation

Syntax-directed code generation is a very simple way of generating machine code from an abstract syntax tree. We traverse the abstract syntax tree recursively, generate code for each child of a node individually and combine these codes with code that comes from the node itself. This technique is very easy to implement because it can be done in a simple recursive AST traversal. However, the quality of the code is usually very low because the code generator lacks a "global view" of the code. To generate high-quality code, more advanced techniques are needed which are out of the scope of this text. Here, we only want to get a first impression of how higher-level languages are implemented with machine code.

To keep our code generator simple, we keep all local variables of a function in its **stack frame** and will use registers only when evaluating expressions. This is of course detrimental to performance because the code will have a lot of loads and stores to the memory but it will keep our code generator simple.

In principle, the code generator works in a similar way as the type elaboration routine that we discussed in the previous section: In the following, we define the following functions:

1. *codeP* generates code for programs.
2. *codeS* generates code for statements.
3. *codeR* generates code the R-evaluation of an expression.
4. *codeL* generates code the L-evaluation of an expression.

We define functions inductively over the abstract syntax of C0.

9.3.1 Expressions

The functions *codeL* and *codeR* generate code for expressions. According to Section 6.4, only two elements of the abstract syntax of C0 are L-evaluable: the occurrence of a variable and the indirection expression. In our compiler, the L-evaluation of an expression yields a memory address. According to Definition 6.3.2 we model the R-evaluation of L-evaluable expression by L-evaluating them and then loading from the resulting address.

The function

$$\mathit{codeR} : (\mathit{Var} \rightharpoonup \mathbb{N}) \longrightarrow \mathit{Regs}^* \longrightarrow \mathit{Expr} \longrightarrow \mathit{Ty} \longrightarrow \mathit{Code}$$

(and *codeL* analogously) takes the following arguments:

1. A mapping that maps each variable's name to its offset in the stack frame.
2. A list of registers that can be used for evaluating the expression.
3. The expression to generate the code for.
4. The type of the expression as determined by the type checker in the last section.

It returns a piece of code the implements the evaluation of the expression. We set up the code such that the resulting value of the expression is always stored in the first register of the register list that we pass to *codeR*/*L*. In the following, we are defining both functions recursively over the different elements of the abstract expression syntax of C0.

Constants. The R-evaluation of a constant just places the constant in the first register of the register list.

$$[\text{CConst}]\ \frac{}{\mathit{codeR}\ \ \mathit{offs}\ \ (r :: rs)\ \ [\![c]\!]\ \ \mathtt{int} = c_{\mathsf{const}}}$$

```
# c_const
li r c
```

Open in Browser

Variables. The L-evaluation of a local variable produces the address of the variable in the stack frame. To this end, we use the function *offs* to lookup the offset d of the variable in the stack frame and add it to the stack pointer. This assumes that the stack frame is sufficiently large. We will ensure this in the code generation for statements by tracking the memory consumption and generating code that allocates a sufficiently large stack frame.

$$[\text{CVar}]\ \frac{\mathit{offs}\, x = d}{\mathit{codeL}\ \ \mathit{offs}\ \ (r :: rs)\ \ [\![x]\!]\ \ k = c_{\mathsf{var}}}$$

```
# c_var
addiu r $sp d
```

Open in Browser

L- to R-evaluation. If an expression is L-evaluable, the R-evaluation of that expression consists of L-evaluating it, which yields the corresponding address and then loading from it. Here, we need the type information to select the appropriate load instruction: When loading a character, we need to load a byte and sign-extend it. When loading an integer or a pointer, we have to load a word. The following rule makes it possible to omit individual R-evaluation rules for L-evaluable expressions and use this rule to handle the R-evaluation for them.

$$[\text{CLtoR}]\ \frac{\begin{array}{c}\mathit{codeL}\ \ \mathit{offs}\ \ (r :: rs)\ \ [\![l]\!]\ \ k = c \\ L = \text{load instruction for type } k\end{array}}{\mathit{codeR}\ \ \mathit{offs}\ \ (r :: rs)\ \ [\![l]\!]\ \ k = c_{LtoR}}$$

```
# c_LtoR
c
L r r
```

Open in Browser

Address-of and Indirection. The operators & and $*$ toggle between L- and R-evaluation. They do however not generate any code themselves!

$$[\text{CIndir}]\ \frac{\mathit{codeR}\ \ \mathit{offs}\ \ (r :: rs)\ \ [\![e]\!]\ \ k* = c}{\mathit{codeL}\ \ \mathit{offs}\ \ (r :: rs)\ \ [\![*e]\!]\ \ k = c}$$

$$[\text{CAddrOf}]\ \frac{\mathit{codeL}\ \ \mathit{offs}\ \ (r :: rs)\ \ [\![l]\!]\ \ k = c}{\mathit{codeR}\ \ \mathit{offs}\ \ (r :: rs)\ \ [\![\&l]\!]\ \ k* = c}$$

Binary Expressions. For binary operators, we show the case for arithmetic operators here, one of the two sub-expressions needs to be evaluated first. The register that was used to keep the value of that sub-expression cannot be used to evaluate the second sub-expression because the register has to hold the computed value throughout the evaluation of the second sub-expression. So, *codeR* needs at least two registers to generate code that evaluates the expression. If the registers are exhausted, the compiler is allowed to report an error and abort compilation.

$$[\text{CBinary}]\ \frac{\begin{array}{c}\mathit{codeR}\ \ \mathit{offs}\ \ (r_1 :: r_2 :: rs)\ \ [\![e_1]\!]\ \ k_1 = c_1\\ \mathit{codeR}\ \ \mathit{offs}\ \ (r_2 :: rs)\ \ [\![e_2]\!]\ \ k_2 = c_2\end{array}}{\mathit{codeR}\ \ \mathit{offs}\ \ (r_1 :: r_2 :: rs)\ \ [\![e_1\ o\ e_2]\!]\ \ k = c_{Binary}}$$

In principle, one could define an additional rule

$$[\text{CBinary2}]\ \frac{\begin{array}{c}\mathit{codeR}\ \ \mathit{offs}\ \ (r_1 :: r_2 :: rs)\ \ [\![e_2]\!]\ \ k_1 = c_2\\ \mathit{codeR}\ \ \mathit{offs}\ \ (r_2 :: rs)\ \ [\![e_1]\!]\ \ k_2 = c_1\end{array}}{\mathit{codeR}\ \ \mathit{offs}\ \ (r_1 :: r_2 :: rs)\ \ [\![e_1\ o\ e_2]\!]\ \ k = c_{Binary2}}$$

which just swaps the order in which the sub-expressions are evaluated as can be seen in the code that they generate:

```
# c_Binary
c1
c2
o  r1 r1 r2
```

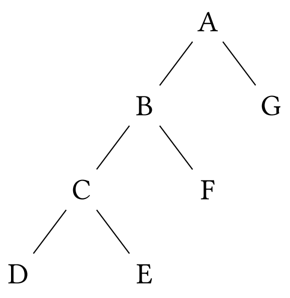
Open in Browser

```
# c_Binary2
c2
c1
o  r1 r1 r2
```

Open in Browser

In principle, having both rules [CBinary] and [CBinary2] would make our code generator non-determinstic: For each binary expression we could apply either rule. So, it would be nice to have a *deterministic* policy that tells us when to use which rule.

Looking at this closer, it turns out that the order in which the expressions are evaluated has an impact on the register consumption. Consider the following expression tree:



When evaluating the right sub-expression first in each expression, the code looks as follows

```
G r1
F r2
E r3
D r4
C r3 r4 r3
B r2 r3 r2
A r1 r2 r1
```

and will use four registers. When generating code for the left sub-expression first in each expression, we get code that looks like this

```
D r1
E r2
C r1 r1 r2
F r2
B r1 r1 r2
G r2
A r1 r1 r2
```

which uses only two registers. Now, fixing the order in which code for the expressions is generated *a priori*, can yield a sub-optimal register consumption. The question is if there is a way of determining the order in which to generate code for the expressions such that register consumption is minimized?

It turns out there is. The following observation is key: Consider a binary expression e. Assume we knew the register consumption of both sub-expressions of e: Say the code for the left sub-expression e_1 uses $regs(e_1)$ registers and the code for the right sub-expression uses $regs(e_2)$ registers. If $regs(e_1) = regs(e_2)$, it doesn't matter which sub-expression we evaluate first because we need one register to keep the result of one sub-expression while evaluating the other. So, in total, we need $regs(e_1) + 1$ registers to evaluate e itself. However, if $regs(e_1) \neq regs(e_2)$, then we evaluate the one with the higher register demand first, and then we can use one of the registers we used to evaluate that sub-expression that won't be used by the other in which we can keep the value. So, in conclusion: We should always generate code for the sub-expression first that has the higher register demand. We can determine the register demand recursively by the following function that implements the considerations above:

$$\begin{aligned} regs(x) &= 1 \\ regs(c) &= 1 \\ regs(e_1\ o\ e_2) &= \begin{cases} regs(e_1) + 1 & \text{if } regs(e_1) = regs(e_2) \\ \max(regs(e_1), regs(e_2)) & \text{otherwise} \end{cases} \end{aligned}$$

One can show [17] that generating code this way results in *minimal* register consumption. In summary, our compiler can use the function *regs* to break the non-determinism of [CBinary] and [CBinary2] and always select the one that leads to minimal register consumption.

9.3.2 Statements

Let us consider code generation for statements which is handled by the function

$$codeS : (Var \rightharpoonup \mathbb{N}) \to Stmt \to Code$$

which takes the following parameters:

- A mapping that maps each variable's name to its offset in the stack frame.
- A C0 statement.

codeS returns a piece of MIPS code that implements the statement.

While and If. The code of the while loop and the if-then-else is composed from the code that implements the bodies of the statements and the code that implements the R-evaluation of the conditional. Note that the labels in the machine code have to be unique, i.e. for each instance of a while loop or an if-then-else in the program, we have to use *fresh* label names.

$$[\text{CWhile}]\ \frac{\mathit{codeR}\ \ \mathit{offs}\ \ (r :: rs)\ \ [\![e]\!]\ \ k = c_1 \qquad \mathit{codeS}\ \ \mathit{offs}\ \ [\![s]\!] = c_2}{\mathit{codeS}\ \ \mathit{offs}\ \ [\![\texttt{while}\,(e)\,s]\!] = c_{\mathsf{While}}}$$

```
    # c_while
    b T
L:
    c2
T:
    c1
    bnez r L
```

Open in Browser

$$[\text{CIf}]\ \frac{\mathit{codeR}\ \ \mathit{offs}\ \ (r :: rs)\ \ [\![e]\!]\ \ k = c \qquad \mathit{codeS}\ \ \mathit{offs}\ \ [\![s_1]\!] = c_1 \qquad \mathit{codeS}\ \ \mathit{offs}\ \ [\![s_2]\!] = c_2}{\mathit{codeS}\ \ \mathit{offs}\ \ [\![\texttt{if}\,(e)\,s_1\,\texttt{else}\,s_2]\!] = c_{\mathsf{If}}}$$

```
    # c_if
    c
    beqz r F
    c1
    b N
F:
    c2
N:
```

Open in Browser

Assignment. For the assignment, we have to evaluate the left-hand and right-hand side expressions. The evaluation of the left-hand side yields an address and the right-hand side a value. After both expressions have been evaluated, we need to store the value into the stack slot at the address we got. To this end, we need the appropriate store that stores a value of type k_1 which is the type of l. Note also that we have freedom in which sub-expression (left or right) we evaluate first and the same reasoning as for the binary arithmetic expression above applies here. For the sake of brevity we are only giving one rule here.

$$[\text{CAssign}]\ \frac{\mathit{codeL}\ \ \mathit{offs}\ \ (r_1 :: r_2 :: rs)\ \ [\![l]\!]\ \ k_1 = c_1 \qquad \mathit{codeR}\ \ \mathit{offs}\ \ (r_2 :: rs)\ \ [\![e]\!]\ \ k_2 = c_2 \qquad S = \text{store that stores value of type } k_1}{\mathit{codeS}\ \ \mathit{offs}\ \ [\![l = e;]\!] = c_{\mathsf{Assign}}}$$

```
# c_assign
c1
c2
S r2 (r1)
```

Open in Browser

Blocks. The code for a block *is* the code for the constituent program. We discuss code generation for programs in Subsection 9.3.3. In addition, the block rule takes care of allocating stack slots for the local variables declared in the block. The number δ is the maximum offset assigned in the current stack slot assignment *offs*. The code for the constituent statements is generated with an extended stack slot assignment *offs′* that has room for the local variables declared in the block.

$$[\text{CBlock}]\ \frac{\delta = \max\{\mathit{offs}(x) \mid x \in \operatorname{dom} \mathit{offs}\} \qquad \mathit{offs}' = \mathit{offs}[x_i \mapsto \delta + 4i \text{ for } 1 \le i \le m] \qquad \mathit{codeP}\ \ \mathit{offs}'\ \ [\![p]\!] = c}{\mathit{codeS}\ \ \mathit{offs}\ \ [\![\{k_1\ x_1; \dots; k_m\ x_m;\ p\}]\!] = c_{\mathsf{Block}}}$$

```
# c_block
c
```

Open in Browser

9.3.3 Programs

Generating code for the program part of a C0 program is simple. The empty program has no associated code. The sequence operator just prepends the code generated for the statement to the code generated for the program.

$$[\text{CTerm}]\ \frac{}{\mathit{codeP}\ \ \mathit{offs}\ \ [\![\varepsilon]\!] = c_{\mathsf{Term}}}$$

```
# c_term
```

Open in Browser

$$[\text{CSeq}]\ \frac{\mathit{codeS}\ \ \mathit{offs}\ \ [\![s]\!] = c_s \qquad \mathit{codeP}\ \ \mathit{offs}\ \ [\![p]\!] = c_p}{\mathit{codeP}\ \ \mathit{offs}\ \ [\![s\ p]\!] = c_{\mathsf{Seq}}}$$

```
# c_seq
c_s
c_p
```

Open in Browser

9.3.4 The Function Prologue and Epilogue

We'll use the code generation functions for expressions and statements that we have devised in the previous two subsections to generate code for an entire function. Our formalization of C0 does not know functions but we are discussing a small extension of the language here nevertheless. The code c of the body p of a function f with parameters $k_1\,x_1; \dots; k_m\,x_m;$ can be generated by considering the function's body wrapped in a block where parameters are declared as local variables:

$$c = \mathit{codeS}\ \ \emptyset\ \ [\![\{k_1\,x_1; \dots; k_m\,x_m;\ p\}]\!]$$

The body code c now has to extended by a **prologue** and **epilogue** that sets up the stack frame and saves the return register:

```
f:
    .globl
    subiu $sp $sp S+4
    sw    $ra S($sp)
    sw    $a0 0($sp)
    ...
    sw    $an 4n($sp)
    c     # body code
f_end:
    lw    $ra S($sp)
    addiu $sp $sp S+4
    jr    $ra
```

Open in Browser

For the sake of simplicity, we are only considering functions that have no more than four parameters which is the maximum number of parameters passed in registers. Functions with more parameters use the stack to pass the additional arguments which would complicate our code slightly. S is the maximum size of the stack frame, i.e. the maximum offset used in rule [CVar] plus 4. Here, the stack frame is four more bytes large because we also have to save the return address. Before placing the body code c, we save the argument registers into the stack frame so that they can be accessed like local variables.

Appendix A

MIPS Assembler Reference

A.1 Instruction Set Reference

Here, we summarize the MIPS instructions and OS calls that are most important for this book. The official MIPS reference [3] provides further details on the MIPS-IV instruction set.

Unless stated otherwise, each instruction increments the program counter by 4. We use $\mathrm{se}(b)$ to denote a sign extension of a bit string to 32 bit and $\mathrm{ze}(b)$ for a zero extension of a bit string to 32 bit (see Sign and Zero Extension). Note that instructions using an immediate perform either sign or zero extension of the 16-bit immediate to 32 bit depending on the opcode. $<_s$ und $<_u$ denote signed and unsigned comparisons. $W[a]$, $H[a]$ und $B[a]$ denote memory contents at address a in form of a word, half word or byte.

We use the following abbreviation to compute the new pc address for a conditional jump:

$$cbr(cc) \coloneqq pc + 4 \cdot (1 + \mathit{if}\ \ cc\ \ \mathit{then}\ \mathrm{se}(i)\ \ \mathit{else}\ 0)$$

Mnem	Args	Semantics	Comment
addu	$\$d\ \$s\ \$t$	$\$d \leftarrow \$s + \$t$	Add two regs
addiu	$\$d\ \$s\ i$	$\$d \leftarrow \$s + \mathrm{se}(i)$	Add reg with immediate
subu	$\$d\ \$s\ \$t$	$\$d \leftarrow \$s - \$t$	Subtract two regs
mul	$\$d\ \$s\ \$t$	$\$d \leftarrow \$s \cdot \$t$	Multiply two reg
div	$\$d\ \$s\ \$t$	$\$d \leftarrow \$s\ /\ \$t$	Divide two regs
rem	$\$d\ \$s\ \$t$	$\$d \leftarrow \$s \,\%\, \$t$	Remainder of division
and	$\$d\ \$s\ \$t$	$\$d \leftarrow \$s\ \&\ \$t$	Bitwise and
andi	$\$d\ \$s\ i$	$\$d \leftarrow \$s\ \&\ \mathrm{ze}(i)$	Bitwise and with immediate
or	$\$d\ \$s\ \$t$	$\$d \leftarrow \$s \mid \$t$	Bitwise or
ori	$\$d\ \$s\ i$	$\$d \leftarrow \$s \mid \mathrm{ze}(i)$	Bitwise or with immediate
xor	$\$d\ \$s\ \$t$	$\$d \leftarrow \$s\ \hat{}\ \$t$	Bitwise xor
xori	$\$d\ \$s\ i$	$\$d \leftarrow \$s\ \hat{}\ \mathrm{ze}(i)$	Bitwise xor with immediate
nor	$\$d\ \$s\ \$t$	$\$d \leftarrow \overline{\$s \mid \$t}$	Complement of or
lui	$\$d\ \ i$	$\$d \leftarrow i_{15} \ldots i_0\, 0^{16}$	Load upper half-word of reg
sll	$\$d\ \$s\ n$	$\$d \leftarrow \$s_{31-n} \ldots \$s_0\, 0^n$	Shift left by n
srl	$\$d\ \$s\ n$	$\$d \leftarrow 0^n\, \$s_{31} \ldots \$s_n$	Unsigned shift right by n
sra	$\$d\ \$s\ n$	$\$d \leftarrow (\$s_{31})^n\, \$s_{31} \ldots \$s_n$	Signed shift right by n
sllv	$\$d\ \$s\ \$t$	$\$d \leftarrow \$s_{31-n} \ldots \$s_0\, 0^n$	$n := \$t$
srlv	$\$d\ \$s\ \$t$	$\$d \leftarrow 0^n\, \$s_{31} \ldots \$s_n$	$n := \$t$
srav	$\$d\ \$s\ \$t$	$\$d \leftarrow (\$s_{31})^n\, \$s_{31} \ldots \$s_n$	$n := \$t$
slt	$\$d\ \$s\ \$t$	$\$d \leftarrow \mathit{if}\ \$s <_s \$t\ \mathit{then}\ 1\ \mathit{else}\ 0$	Signed comparison of two regs
sltu	$\$d\ \$s\ \$t$	$\$d \leftarrow \mathit{if}\ \$s <_u \$t\ \mathit{then}\ 1\ \mathit{else}\ 0$	Unsigned comparison of two regs
slti	$\$d\ \$s\ i$	$\$d \leftarrow \mathit{if}\ \$s <_s \mathrm{se}(i)\ \mathit{then}\ 1\ \mathit{else}\ 0$	Signed comparison of reg with imm.
sltiu	$\$d\ \$s\ i$	$\$d \leftarrow \mathit{if}\ \$s <_u \mathrm{se}(i)\ \mathit{then}\ 1\ \mathit{else}\ 0$	Unsigned comparison of reg with imm.
lw	$\$d\ i(\$s)$	$\$d \leftarrow W[\$s + \mathrm{se}(i)]$	Load word
lh	$\$d\ i(\$s)$	$\$d \leftarrow \mathrm{se}(H[\$s + \mathrm{se}(i)])$	Load half word with sign extension
lb	$\$d\ i(\$s)$	$\$d \leftarrow \mathrm{se}(B[\$s + \mathrm{se}(i)])$	Load byte with sign extension
lhu	$\$d\ i(\$s)$	$\$d \leftarrow \mathrm{ze}(H[\$s + \mathrm{se}(i)])$	Load half word with zero extension
lbu	$\$d\ i(\$s)$	$\$d \leftarrow \mathrm{ze}(B[\$s + \mathrm{se}(i)])$	Load byte with zero extension
sw	$\$d\ i(\$s)$	$W[\$s + \mathrm{se}(i)] \leftarrow \$d$	Store word
sh	$\$d\ i(\$s)$	$H[\$s + \mathrm{se}(i)] \leftarrow \$d[15:0]$	Store half word
sb	$\$d\ i(\$s)$	$B[\$s + \mathrm{se}(i)] \leftarrow \$d[7:0]$	Store byte
beq	$\$s\ \$t\ i$	$pc \leftarrow cbr(\$s = \$t)$	
bne	$\$s\ \$t\ i$	$pc \leftarrow cbr(\$s \neq \$t)$	
bccz	$\$t\ i$	$pc \leftarrow cbr(\$t\ cc\ 0)$	$cc \in \{\mathrm{lt}, \mathrm{gt}, \mathrm{le}, \mathrm{ge}\}$ (signed comparison)
jal	*addr*	$\$ra \leftarrow pc + 4, pc \leftarrow addr$	Function call
jr	$\$s$	$pc \leftarrow \$s$	Indirect jump
syscall			Invoke operating system
li	$\$d\ i$	$\$d \leftarrow i$	Load constant into reg
la	$\$d\ l$	$\$d \leftarrow addr$	Load address of label into reg
move	$\$d\ \$s$	$\$d \leftarrow \$s$	Copy reg
not	$\$d\ \$s$	$\$d \leftarrow \overline{\$s}$	Bitwise complement
neg	$\$d\ \$s$	$\$d \leftarrow -\$s$	$\cdot(-1)$
b	i	$pc \leftarrow pc + 4 \cdot (1 + \mathrm{se}(i))$	Unconditional jump
bcc	$\$s\ \$t\ i$	$pc \leftarrow cbr(cc)$	$cc \in \{\mathrm{lt}, \mathrm{gt}, \mathrm{le}, \mathrm{ge}\}$ (signed comparison)

A.2 OS Calls

Calling the operating system is done by the `syscall` instruction. The operation to be carried out is specified by the `$v0` register. Potential arguments to the system call are passed in the argument registers.

Action	`$v0`	Description
print integer	1	`$a0` = number to be printed.
print string	4	`$a0` = address of ASCIIZ string.
read integer	5	Read number stored in `$v0`.
read string	8	`$a0` = start address of memory area where characters are put. `$a1` = maximum number of characters to read.
exit	10	End program execution.
print character	11	`$a0` = ASCII code of character to print.
read character	12	ASCII code of read character stored in `$v0`.

A.3 Assembler Directives

Directive	Description
.align n	Align next datum to an address divisible by 2^n
.ascii str	Add ASCII string *str* to data segment
.asciiz str	Add zero-terminated ASCII string to data segment
.byte b1, ..., bn	Add bytes b_1 to b_n successively into segment
.data	Start/continue data segment
.globl sym	Declare symbol *sym* as global (visible from other object files)
.half h1, ..., hn	Add half words h_1 to h_n to segment
.space n	Reserve *n* bytes of free space in current segment
.text	Start code segment
.word w1, ..., wn	Add words w_1 to w_n successively to current segment

Appendix B

ASCII Table

Dec	Hex		Dec	Hex		Dec	Hex		Dec	Hex	
0	00	NUL	32	20		64	40	@	96	60	`
1	01	SOH	33	21	!	65	41	A	97	61	a
2	02	STX	34	22	"	66	42	B	98	62	b
3	03	ETX	35	23	#	67	43	C	99	63	c
4	04	EOT	36	24	$	68	44	D	100	64	d
5	05	ENQ	37	25	%	69	45	E	101	65	e
6	06	ACK	38	26	&	70	46	F	102	66	f
7	07	BEL	39	27	’	71	47	G	103	67	g
8	08	BS	40	28	(	72	48	H	104	68	h
9	09	TAB	41	29	)	73	49	I	105	69	i
10	0A	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	45	2D	-	77	4D	M	109	6D	m
14	0E	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	|
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	DEL

Appendix C

Course Structure

This chapter provides a sample schedule for using this script in a course of one semester with 25 lectures of 90 minutes each.

Week	Lecture 1	Minitest	Lecture 2	Project
1	I/R1		R2	
2	M1	A	M2	
3	M3		C1	M1
4	C2	M	C3	M1
5	A1		A2	C1
6	S1	C	S2	C1
7	S3			C2
8	S4	S	J1	C2
9	J2		J3	J1
10	J4	J	A3	J1
11	A4		O1	J2
12	O2	A	O2	J2
13	V1		V2	J2/3
14		O/V		J3
15				J3

The abbreviations mean:

R	Arithmetic
M	Machine Code
C	C
A	Algorithms
S	Semantics
J	Java
O	Compilers
V	Verification

C.1 I: Course Introduction

Synopsis.

- Introduce the goals of the course and give an overview of its content.
- Introduce the basic modus operandi explaining the course structure, projects, tests, exams, etc.

Comments. Ideally, one does not spend more then 45 minutes for this.

C.2 R1: Computer Arithmetic

Synopsis.

- Recap Positional number systems.
- Understand that sequences of digits are just sequences of symbols and that they require an embedding into the naturals.
- Understand that the positional systems make sense because there is a bijection between digit sequences of length k and naturals between 0 and $2^k - 1$. This is an opportunity to recap simple induction proofs.
- Understand binary and hexadecimal numbers and the appropriate terminology.
- Introduce and, or, xor as binary operations on bits that can be easily implemented in hardware.
- Understand that computers operate on binary numbers of a fixed length and that this permeates the type systems of many imperative programming languages for performance reasons, so eventually people have to be aware of this and the consequences (overflows).

Sections Covered. Section 1.1, Section 1.2

C.3 R2: Computer Arithmetic

Synopsis.

- Introduce the binary adder, make clear that it is just a combinational circuit and prove the addition of individual bits by a value table and prove that addition on bit strings it is sound with respect to the unsigned interpretation.
- Make clear that the exact result needs one bit more than the operands are long and show how to use it to detect overflows.
- Show how we can use modulo arithmetic to implement subtraction using addition.
- Introduce signed numbers and show that addition and subtraction as we defined it is also sound on them.
- Make clear that overflows have to be determined differently on signed numbers.
- Explain sign and zero extension (will be of some importance later on when talking about load instructions in the machine code section.)
- Bonus: Shifts

Sections Covered. Section 1.3, Section 1.4

C.4 M1: Introduction to Machine Code Programming

Synopsis.

- Basic, very high-level understanding of computer architecture.
- MIPS processor, registers, program counter, memory, bus, rest of the system
- What is an instruction, how does it look like?
- Different kinds of instructions (arithmetic, load/store, control).
- Discuss kinds of instructions by means of the simple factorial function code.
- Introduce the assembler, relieves us from coding instructions manually, provides labels for addresses, pseudo instructions, has directives. Introduce separate translation with translation units (will be the same in C and Java and many other compiled languages).
- Execution traces. Execute programs on paper to deepen understanding of program execution.

Sections Covered. Section 2.1, Section 2.2, Section 2.3, Section 2.4, Section 2.7

C.5 M2: Simple Algorithms in Machine Code

Synopsis.

- Practice programming in machine code using simple examples.
- Introduce other aspects on the fly: data segment, strings.
- Live coding two simple examples from Chapter 3.
- The print hex example is useful to recap “bit fiddling”, exercise number systems, and introduce OS calls, strings (ask user for input).
- Horner, Eratosthenes, Tortoise and Hare are useful to practice memory instructions and simple loops.
- The former are useful to discuss array processing.

Sections Covered. Chapter 3

C.6 M3: Function Calls

Synopsis.

- Explain function call mechanism. `jal` and `jr` and the return address.
- Mention the problem that we want to preserve values over calls (first and foremost the return address).
- Such values have to be saved somewhere; registers may be overwritten by called function.
- So we need a convention which registers are preserved and which can be overwritten.

- Overwritten registers must be saved somewhere. Global data not possible because recursive calls would overwrite saved values.
- Need call stack. Explain allocation/deallocation of stack frame, usage of stack pointer. Show how saving/restoring is done (relative addressing to stack pointer.)
- Discuss recursion and tail recursion in this context. Live code factorial recursively and tail recursively. Step through program and show the difference.
- Goal is to understand that recursion requires book keeping and has an overhead.

Sections Covered. Section 2.8

C.7 C1: Introduction to C

Synopsis. This lecture gives an introduction and an overview of the basics of C.

- Motivate why we need a more high-level language (Chapter 4 intro).
- Introduce syntactical and static properties (translation units, functions, declarations, identifiers, keywords, statements, expressions) by means of the factorial example (Section 4.1).
- Discuss dynamics (function calls, statements, loops) also on that example.
- Deepen that understanding by walking through an execution trace (Section 4.2).
- Discuss concept of an imperative variable (in contrast to functional / mathematical variables) (Section 4.3). Stress that containers have an identity (address), an extent, and a life time.
- Give an overview over the C memory model (Section 4.4).
- Translation units, different C files compiled separately, demystify ugly header files and include mechanism. main function and program arguments (Section 4.5).

Sections Covered. Section 4.1, Section 4.2, Section 4.3, Section 4.4, Section 4.5

C.8 C2: Expressions and Pointers

Synopsis.

- Briefly outline C's numeric types (Section 4.6), mention signed and unsigned, and explain `sizeof` properly. `sizeof` is often mistaken for some kind of run time construct that gives the size of a container.
- Overview of C's rich expression language (Section 4.8).
- Implicit type conversion.
- L- and R-evaluation.
- Pointers. Motivation: Passing large amounts of data to function. Don't want to copy. Like hiring a contractor: You don't bring them your house but give them your address. Go through execution trace example. Discuss L- and R-eval rules (Section 4.10).
- Arrays. are special. Degenerate to pointer to first element.

Sections Covered. Section 4.6, Section 4.8, Section 4.10

C.9 C3: Arrays, Structs, I/O

Synopsis.

- Discuss basic array handling by means of a small example: read in numbers from the console and put them into an array and compute the average/sum/something similar.
- Understand that arrays degenerate to pointer of first element in the context of R-eval.
- Emphasize that `sizeof` is the size of the type and evaluated at compile time. Cannot use it to query size of container at runtime. Makes a difference for malloc'ed containers.
- Start a more complex example. Read in person records from a file (name, surname, date of birth)
- Discuss basic I/O `printf`, `fscanf`, `fopen`, `fclose`.
- Struct become helpful. Discuss composition vs aggregation. Mention that strings are null-terminated arrays of characters.
- Simplify program with respect to pointer notation; explain `->` and `.`.
- Modularize program into different files, creating functions for reading and printing records.
- Add command line argument for file name and explain `argc` and `argv`.

Sections Covered. Subsection 4.10.1, Section 4.13, Section 4.14

C.10 A1: Lists and Trees

Synopsis.

- Start with the example from last time (reading in person records).
- Show that reading into a local array is not nice because we are limited in the amount of records to read. Alternatives may be reading file twice or starting file with number of records. We don't want that.
- Need list of records that grows while reading: array list using realloc. Discuss amortized complexity of appending.
- Alternative linked list. Discuss and show implementation.
- Summarize pros and cons of array lists vs linked lists.
- Discuss trees. Hierarchical data structure. Mostly used to implement sets or maps.
- Basic property: Set of nodes where each node is linked to multiple children but each node has at most one parent.
- Discuss basic notions: labels/keys, parent, child, root, graph of a tree, size, height, depth and present height computation as a simple recursive algorithm.

- Discuss search trees and search tree property. Show searching and insertion. Discuss complexity of search in balanced trees. Of course, insertion may violate balancedness.
- Introduce tries as maps where keys are sequences of things.
- Present searching and inserting in tries.

Sections Covered. Section 5.1, Section 5.2

C.11 A2: Dynamic Programming

Synopsis.

- Dynamic programming is a technique to solve optimization problems that have optimal substructure
- It uses memoization/tabulation to solve problems with overlapping sub-problems more efficiently
- Intro example is shortest path; demonstrate that longest simple paths doesn't have optimal sub-structure
- Explain memoization by Fibonacci numbers or binomial numbers
- Introduction edit distance as an optimization problem and go through example.

Sections Covered. Subsection 5.4.1 or Subsection 5.4.2, Subsection 5.4.3

C.12 S1: C0 Syntax and Semantics

Synopsis.

- Introduce syntax definitions, understand difference between abstract and concrete syntax.
- Introduce C0 statement and expression language.
- Structure of a C0 state.
- Expression evaluation semantics
- Small-step operational semantics for statements.
- Different modes in which program behave: termination, divergence, aborting, getting stuck.

Sections Covered. Section 6.1, Section 6.2, Section 6.3

C.13 S2: Extending C0 to Pointers and Scopes

Synopsis.

- So far, C0 cannot declare local variables and has no pointers.
- First introduce pointers and recap L- and R-evaluation formally.
- Second, add scopes by extending the block syntax and changing the state to a stack of variable assignments.
- Introduce and discuss new block and leave rules that administer the stack of variable assignments.

Sections Covered. Section 6.4, Section 6.5

C.14 S3: Correctness

Synopsis.

- Introduce specifications semantically as pair of state sets.
- Show examples of programs that fulfill simple specifications.
- Introduce assertions as propositional formula that represents sets of states and can be used to formulate specifications by pre- and post-conditions.
- Define specifications syntactically pre- and post-conditions and solve the technical problem of compatible variables.
- Introduce total and partial correctness and the syntax of Hoare triples.
- Discuss weakening of post-conditions and strengthening of pre-conditions.
- Introduce failures, discuss difficulties in detecting failures in C due to undefined behavior.
- Introduce assertions practically as a means to check if an assertion holds during program execution.
- Briefly discuss “defensive programming”, i.e. making programs fail as early as possible by documenting pre- and post-conditions using assertions.

Sections Covered. Section 7.1

C.15 S4: Testing

Synopsis.

- Failures witness bugs.
- Failures are symptoms no diagnosis; failure and error location often far apart.
- Use assertions to document specifications as invariants to move failures closer to error.
- Program defensively, don’t just consider the happy path.

- Introduce tests as states in theory and small programs that set up states in practice.
- Black-box tests are created based on a specification only but cannot guarantee any coverage of the tested code. Different tests test different "cases" in the specification.
- White-box tests test existing code and try to reach high coverage but fail to detect missing code.
- Coverage is measured in different criteria: Branch, path, statement coverage.

Sections Covered. Section 7.2, Section 7.3

C.16 J1: Intro to Java

Synopsis.

- Start with comparison of C and Java.

C	Java
struct	class
container	object
function	method
pointer	reference
malloc	new
free	garbage collection (no leaks, no dangling pointers)
unsigned	—
undefined behavior	—
sanitizer	totalized semantics

- Outline basic compilation process. .java → .class → JVM
- Overview over Types. Base types vs reference types.
  - Cannot take address of base type variables, cannot allocate them on the heap, lifetime determined by scopes.
  - No * and &.
  - Reference types are classes and arrays.
  - Allocated with new, life time governed by garbage collection.
  - A reference either contains "null" or references a live object.
- Intro arrays.
  - Array contains elements *and* length.
  - You can query the length with ".length".
  - Cannot increase the length, use "ArrayList" for array lists.
  - No multi-dimensional arrays.
  - String is a class that consists of character array.
- First simple example: HelloWorld, compile manually with "javac" and run with JVM "java".
- Switch to IDE. Start with "Vec2" example. Use C program to motivate classes.

  - Proper way of encapsulation in C: Header file with anonymous struct, set of functions, first parameter is always ptr to struct that represents data structure.
  - In Java, you have more syntactic sugar. You can put functions/methods inside the class, "this" is implicit first parameter: reference to object of class.
  - Objects need to be constructed: Introduce constructors, default constructors.
  - Add main method and run example.
  - Add "Rectangle" class. How does it interact with "Vec2"? Encapsulate fields using getters and setter. Motivation: Can change internal implementation of "Vec2" to polar coordinates.
- Show that setters are also a good means of asserting class invariants (Fraction example, `setDenominator`).

Sections Covered. Section 8.1, Section 8.2, Section 8.3, Section 8.4

C.17 J2: Immutable Objects, Interfaces, Subtyping

Synopsis.

- Start with a discussion of immutable vs mutable object.
  - Immutability allows objects to be used like values
  - Effects of shared ownership through aliasing are hard to control.
  - Work through immutable objects with the Vec2 example.
  - Discuss factory methods in light of this example: Need a way to make a Vec2 by cartesian and by polar coordinates. Same arguments, cannot overload the constructor. Static factory method is nice because we can also give it a proper name.
- Motivate inheritance with Vec2: Why only cartesian *or* polar.
- We want to have both. But typing forces us to be specific when using a Vec2, doesn't it? We don't want to be specific with respect to the concrete type, but just say, we expect something that is a Vec2.
- Java has interfaces to specify that. They introduce a *subtyping* relation.
- Introduce Vec2 interface and implement classes PolarVec2 and CartesianVec2. Demonstrate how the type system enforces that all methods required by the interface are implemented in the classes. Show how the Rectangle class can "retreat" to Vec2 and use Polar and Cartesian Vec2's interchangeably.
- Introduce dynamic dispatch. There are two types: The static type of a reference and the dynamic type. The dynamic type is a property of a program execution: It is the type the object was constructed from which is evident at the constructor call. The static type is an upper bound to each dynamic type: "If the static type is T, in every execution, the referenced object is a subtype of T". The method to call is determined by the dynamic type and selected at the run time of the program. The Java type system ensures that this method exists.

Sections Covered. Section 8.6, Section 8.7

C.18 J3: Abstract Classes, `Object`, Overriding and Overloading Methods

Synopsis.

- Motivate inheritance of methods by code reuse by means of the Vec2 example.
  - Multiple variants share code. But code duplication is bad.
  - The `translate` method is implemented in both classes. Factor this method out into a super class `BaseVec2`.
  - However, `setXY` is not available in `BaseVec`.
  - Make `BaseVec2` abstract and declare method `setXY` abstract in `BaseVec2`.
  - Subclasses can always override methods. `CartesianVec2` could override `translate` and provide a more "direct" implementation.
  - Mention `@override`. Defer discussion to overloading.
- `Object`
  - `toString` used to automatically convert an object to a string. Used by the `+` operator on strings and many other methods such as `System.out.println`.
  - `equals()` compares objects to be equal (not the same; that is done by the == operator!). We need this if we want to put objects in collections. Make small example, comparing two strings. Implement in `BaseVec2`.
  - `equals(Vec2)` would be nice but does not override `equals(Object)` that we inherit from `Object`. `@override` helps to make clear that we are actually overriding a method. Will cause compile error if we are overloading instead of overriding.
  - We cannot call `getX` from `equals(Object)` because the static type is too weak. `instanceof` allows for tests on the dynamic type and guard dynamic type casts.
  - Illustrate subtype relationship by means of a Venn diagram. Make clear that we can always weaken the static type. Strengthening it needs a dynamic cast. Draw connection to assertion lecture: weakening post-condition is always possible.
  - Using `instanceof` is bad style and hints at a problem in the design. Sometimes, like in `equals` it is however necessary. Work as much as possible with static types.
- Overloading.
  - Define signature of method with a certain name: list of parameters including implicit `this` argument. Return type is not part of signature.
  - Overloading means that we have multiple methods with the same name but different signature.
  - Overloaded method is identified at *compile type* based on the *static types* of the method's arguments.
  - In contrast to overriding: identifies method to call at *run time* based on the *dynamic types* of the the first (`this`) argument.
  - Identification of suitable overloaded method for a call:
    1. Identify set of methods with appropriate number of parameters.
    2. From that, identify set of applicable methods; if set empty: error.
    3. From that set, identify most specific method; if not unique: error.

Sections Covered. Section 8.7, Section 8.8

C.19 J4: Using OO Properly, OO in C, The Expression Problem, Visitor Pattern

Synopsis. This lecture shall consolidate the understanding of OO software (design) by giving different perspective. We do this in three parts.

1. We introduce a small example (abstract syntax of a small expression language) and model the data structure in a functional, imperative, and OO language. Functional languages have algebraic data types and matching that make the life of the programmer easy. In C one needs to literally implement tagged unions and do the case distinctions on a very low level. In OO, each variant becomes a class and the name of the ADT is typically an interface. The cases of the functional and imperative programs become individual methods in the respective classes.
2. We discuss how virtual method calls are actually implemented by looking at a corresponding implementation in C. The important thing here is to understand that the address of the vtab corresponds to the dynamic type of the object. Instead of doing a case distinction as in (1), we use a double indirection to directly (and efficiently) jump to the right code.
3. The Expression Problem shows the difference of the case-distinction style and OO style when *extending* a class hierarchy. Adding new variants, can be locally done in OO but requires global changes in case-distinction languages. Adding new functionality can be done locally in case-distinction approaches but require global changes in OO. The visitor pattern can help here because it allows for "factoring out" the extension point. Here, understanding the interplay between overriding and overloading is intricate and crucial and can help to understand these two techniques in general.

Sections Covered. Section 8.10, Section 8.11, Section 8.12

C.20 A3: Java Collections Framework and Hashing

Synopsis.

1. Short Overview over the Java Collection Framework
   - The Java Collections Framework is a library of data structures. Some we have discussed in Chapter 5
   - Discuss the basic interfaces `List`, `Set`, `Map`
   - Java has special syntax for Iterators, briefly discuss the Iterator interface and explain for loop.
2. Hashing.
   - Discussed binary search trees as an implementation for maps/sets.
   - Recap that implementing maps subsumes sets
   - New idea that is popular in practice: Use function to map keys to array indexes.

- Works well if function is injective: Hardly the case in practice because key set is too large and would be dependent on array size.
- Non-injective function means that we have collisions.
- First idea: Use collision chains/lists to list all collisions per bucket.
- Discuss simple code to search/add in a hash table with separate chaining.
- Discuss the hashCode/equals problem. Refer to Java Collections Framework.
- Introduce load factor. If load factor less than one and hash function scatters elements evenly, time for search constant.
- Needs however resizing of hash table upon insertion based on a threshold.
- Resizing of hash table requires re-hashing: keys can end up in different bins.
- Mention problem of mutability: Mutating objects such that their hash value changes after they have been inserted may make them "vanish".
- Introduce probing. Chaining conceptionally nice but linked lists require more memory and provoke not so local accesses; bad for the cache.
- New hash function that takes hash value and index into collision chain: determines array index.
- Linear probing: simple, contiguous accesses but easy cluster formation.
- Quadratic probing: more complex, not so contiguous but maybe less cluster formation. Interesting question if the it fully utilizes table, i.e. if probing hash function is surjective on array. Yes, if $c = d = 1/2$ and table size is power of two.
- Final example with all three presented techniques.

Sections Covered. Section 8.9, Section 5.3

C.21 A4: Project Specific

Synopsis. In this slot, we cover project specific material. The second Java project is a bit more involved. For example, we have a route planner or a ray tracer. So in this slot, one can discuss algorithmic background for this project such as Dijstra's algorithm for route planning or bounded volume hierarchies for ray tracing.

C.22 O1: Compiler Introduction (Syntax Analysis, Static Semantics)

Synopsis.

- Program is just a stream of characters
- Lexer forms "words" (called tokens) from characters and can already detect lexical errors.
- Parser checks if token stream adheres to concrete syntax. If yes, it builds the AST, if not, it reports errors.
- However, not all syntactically correct programs are valid C programs.
- They can violate static semantics: types don't match.

- Analyzing static semantics ("semantic analysis") can detect such errors and discard erroneous programs.
- As a side effect, it elaborates types which we need later on for code generation.
- Language is statically type-safe if well-typed programs don't get stuck.
- Discuss examples of C programs that are well-typed but get stuck and discuss how Java solves these problems.
- Introduce type system for C0 (Section 6.6): relation of a type environment, an expression, and its type.
- Define typing rules inductively over the syntax.

Sections Covered. Section 9.1, Section 6.6

C.23 O2: Type Checking Examples, Implementation of Type Checking

Synopsis.

- Work through a couple of type checking examples to discuss how our typing rules work.
- Discuss pros and cons of type systems (not in the lecture notes) by means of statically and dynamically typed languages. Statically typed language typically require more programming effort to get the types right. One can generally generate faster code from them, because types don't have to be checked at runtime. With dynamically-typed languages you typically get your code with less hassle because you don't need to care about making the code type. However, you might spend that time of fixing bugs that typing would have prevented. Dynamically-typed languages typically run slower because type checking is performed at runtime. This can be to some extent alleviated by modern just-in-time compilers.
- Outline implementation of type checking in our small compiler.
- For most language elements this is just bottom-up (recursive) tree traversal.
- Interesting are variables occurrences and declarations.
- Need appropriate implementation of type environment.
- Use a stack of maps. Whenever entering a block, push new map with variable names to declaration AST nodes to it. Point to parent environment. When looking up a variable, traverse stack of type environments from top to bottom and look for variable in the respective map.

Sections Covered. Section 6.6, Section 9.2

C.24 O3: Syntax-Directed Code Generation

Synopsis.

- Simple code generation technique that generates very inefficient code.
- Syntax-driven: compose code of an AST node from code of its children and "glue code"
- Three code-gen functions: L-eval, R-eval, and statements.
- All local variables are kept on the stack for simplicity.
- Sethi-Ullman numbering can be used to decide non-determinism in binary operators based on register demand.

Sections Covered. Section 9.3

C.25 V1: Hoare Logics

Synopsis.

- Recap of correctness definition Section 7.1
- Correctness up to now was defined on the semantics, on executions of the program.
- We now develop a logic to reason about correctness without reasoning about executions.
- Note that the rules of the logic have to be proven correct but that is out of scope.
- Introduce basic rules of abort and assignment and informally reason about correctness.
- Block and sequence rules show how free assertions are determined by other rules (assignment).
- If-then-else example shows that sometimes assertions don't "add up".
- Introduce rule of consequence to strengthen preconditions and weaken postconditions.
- Note that non-matching assertions can be dealt with by the consequence rule which spins off *verification conditions*

Sections Covered. Subsection 7.4.1

C.26 V2: Loop Invariants and Mechanizing Verification

Synopsis.

- Introduce loop invariants as assertions that hold before and after loop body.
- Reason about correctness of while loop informally.

- Note that loop invariants have to be provided by the programmer.
- As a simple example prove correct the division algorithm.
- Introduce pc and vc functions to generate verification conditions.
- Show their correctness for while and if.
- Demonstrate verification condition generation for division algorithm.

Sections Covered. Subsection 7.4.1, Subsection 7.4.2

Appendix D

List of Symbols

Symbol	Description	Page
$+_n$	lower n bits of binary addition	7
$-_n$	lower n bits of binary subtraction	7
$a \equiv b \mod n$	a equivalent b modulo n:	
$\exists z \in \mathbb{Z}. a - b = z \cdot n$	7	
$p +\!\!+ p'$	Concatenation of two C0 programs.	129
$\operatorname{dom} f$	The domain (preimage) X of a function $f : X \longrightarrow Y$	134
$\operatorname{img} f$	The image of a function $f : X \longrightarrow Y$: $\{f(x) \mid x \in X\}$	134
$f\vert_X$	Function f restricted to domain X	134
$Q[e/x]$	e replaces x in Q	153

Index

Literature

[1] ISO, *ISO/IEC 9899:1999 Draft, ISO C Standard 1999*, http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf (1999)

[2] Xi Wang and Haogang Chen and Alvin Cheung and Zhihao Jia and Nickolai Zeldovich and M. Frans Kaashoek, *Undefined Behavior: What happened to my code?, Asia-Pacific Workshop on Systems, APSys '12, Seoul, Republic of Korea, July 23-24, 2012* (2012) http://doi.acm.org/10.1145/2349896.2349905 10.1145/2349896.2349905

[3] Charles Price, *The MIPS IV Instruction*, (1995) http://www.cs.cmu.edu/afs/cs/academic/class/15740-f97/public/doc/mips-isa.pdf

[4] Thomas H. Cormen and Charles E. Leiserson and Ronald L. Rivest and Clifford Stein, *Introduction to Algorithms*, MIT Press; (2009)

[5] Nielson, Hanne Riis and Nielson, Flemming, *Semantics with applications: a formal introduction*, (1992) John Wiley & Sons; https://www.cs.ru.nl/~herman/onderwijs/semantics2019/wiley.pdf

[6] Robert Harper, *Practical Foundations for Programming Languages*, (2013) Cambridge University Press; https://www.cs.cmu.edu/~rwh/pfpl/2nded.pdf

[7] Glynn Winskel, *The Formal Semantics of Programming Languages: An Introduction*, (1993) MIT Press;

[8] Michael Norrish, *C formalized in HOL*, (1998) https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-453.pdf

[9] Robert Krebbers, *The C standard formalized in Coq*, (2015) https://robbertkrebbers.nl/thesis.html

[10] Xavier Leroy and Sandrine Blazy, *Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations*, J. Autom. Reasoning **41** no. 1 1-31. (2008) https://xavierleroy.org/publi/memory-model-journal.pdf

[11] Benjamin Pierce, *Types and Programming Languages*, (2002) MIT Press; https://mitpress.mit.edu/books/types-and-programming-languages

[12] Gosling, James and Joy, Bill and Steele, Guy and Bracha, Gilad, *Java(TM) Language Specification, The 3rd Edition*, (2005) 0321246780 Addison-Wesley; http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html

[13] Mauro Pezzè and Michal Young, *Software Testing and Analysis: Process, Principles and Techniques*, Wiley; (2007)

[14] Philip Wadler, *The Expression Problem*, (1998) https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt

[15] Stefan Richter and Victor Alvarez and Jens Dittrich, *A Seven-Dimensional Analysis of Hashing Methods and its Implications on Query Processing*, (2015) Proceedings of the VLDB Endowment **9** no. 3 96-107. http://www.vldb.org/pvldb/vol9/p96-richter.pdf

[16] Reinhard Wilhelm and Helmut Seidl and Sebastian Hack, *Compiler Design - Syntactic and Semantic Analysis*, (2013) Springer; 10.1007/978-3-642-17540-4

[17] Ravi Sethi and Jeffrey Ullman, *The Generation of Optimal Code for Arithmetic Expressions*, (1970) Journal of the ACM **17** no. 4 715-728. ACM; 10.1145/321607.321620