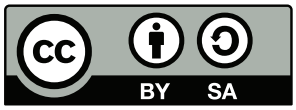
UNIWERSYTET GDAŃSKI
Wydział Matematyki, Fizyki i Informatyki

Maciej Godek
nr albumu: 181131

Data Structure Optimization for Functional Programs

Praca magisterska na kierunku:
INFORMATYKA
Promotor:
dr hab. prof. UG Christoph Schwarzweller

Gdańsk 2017



Abstract

The purpose of this work is to develop techniques to allow for executing programs written in functional style effectively. The work consists of two parts. The first one shows some classic techniques for transforming functional programs into imperative form, as well as some basic methods of proving statements about program properties. In the second part, a method for transforming a certain class of programs operating on lists into equivalent programs operating on arrays is proposed. Furthermore, the conditions allowing to transform a functional implementation of quick sort algorithm into an optimal imperative form are analyzed.

All source programs and transformations are expressed using the purely functional subset of the algorithmic language Scheme, as described in chapter 2. The target computation model is a variant of the RAM machine, whose model and instruction set were described in depth in chapter 3, including an implementation, which uses some imperative features of the Scheme programming language.

In chapter 4 some classic techniques of transforming programs expressed in the previously described subset of Scheme into sequences of instructions for the RAM machine are presented; in particular, the conversion to Continuation-Passing Style and Tail-Call Optimization are described.

Chapter 5 describes a simplified variant of the Boyer-Moore system, including a full list of axioms used for proving theorems about programs expressed in the previously described subset of the Scheme programming language. Unlike the original Boyer-Moore system, however, the system elaborated in our work is incapable of proving theorems on its own, and can only serve as a proof-checker for the proofs provided by its user.

In chapter 6 an original method for converting functional programs into forms receiving and passing arrays is developed. The source language is the purely functional subset of Scheme described in chapter 2, and the target language is the full Scheme language, including its imperative features. The proposed conversion method is only sketchy and certainly requires elaboration.

Chapter 7 deals with automatic conversion of a functional variant of the quick sort algorithm into an imperative form, although it fails to present a working conversion algorithm.

Keywords

data structure, program transformation, compiler, theorem prover, functional programming

Contents

1

Introduction

> *"When I use a word," Humpty Dumpty said, in rather a scornful tone, "it means just what I choose it to mean—neither more nor less."*
>
> – Lewis Carroll, *Through the Looking Glass*

1.1 Motivation

Functional programming is a valuable technique for building large and reusable software systems. Contrasted with more widespread approaches, its main advantage is that it simplifies the reasoning about program's behavior. Furthermore, it allows programmers to provide less detail about the program, making it susceptible to run in various different setups.

1.1.1 Historical perspective – algorithms

Historically, programmers had to specify computations in terms of actions that were to be taken in order to achieve a desired goal. Initially, specifications were written in languages specific to particular machines being programmed.

This style of thinking – i.e. specifying a series of steps to achieve a certain goal – gave birth to the branch of Computer Science known as Algorithm Design.

Being very diverse, machine languages turned out to be inappropriate for communicating algorithms between programmers of different machines. For this reason, the FORTRAN and ALGOL families of languages were invented as means of conveying the ideas of specific algorithms in a precise and uniform manner[71]. In addition, tools existed that allowed to run programs written in those languages on real machines, although with some performance penalty – therefore, for performance-critical applications, programmers were still writing machine code[79].

What those languages had in common is that their main purpose was to instruct computer how to perform computations. In addition, the means of expression of those languages were designed, so that they could easily be translated directly to machine code of most CPUs. With time, however, the way of thinking about computers had changed significantly. No longer were they just mere tools for performing scientific computations, but they were becoming larger and larger systems that were expected to run multiple programs simultaneously in real time.

Furthermore, the complexity of the applications was only increasing, and the old ways of thinking were often insufficient for managing that complexity.

As the performance of computers was improving, our requirements with regard to programming systems begun to shift. For many applications it was no longer necessary to get the maximum performance of every CPU cycle, as there were other factors, such as network latency, that bounded the overall performance of the application.

Moreover, there turned out to be a physical limit on the clock speed of a single CPU core, which caused hardware manufacturers to focus on delivering processors with more and more cores of the same speed. Consequently, the ability to disperse execution of a program on many cores, or even many machines, can often be more profitable than being able to perform a few more instructions per second by a sequential program.

The abundance of processing power had yet another consequence. As programmers no longer had to worry (too much) about computational resources, they could experiment more with the means of expression of their programming languages. After all, it turned out that the two most costly aspects of computer application development were (1) programmers' time and (2) programmers' mistakes. It therefore seems prudent to focus on shortening the time of application development and on minimizing the possibilities of making mistakes.

1.1.2 Programming as expressing ideas

Eventually, the goal of programming is no longer bending computer behavior to programmer's will, but describing that will accurately, precisely and unambiguously.

This is an interesting process: programmers experiment with programming languages, which in turn provide them with means to think about various phenomena, or prompt them to come up with certain ideas. There is a harmful byproduct of this process though, namely – the multitude of programming languages available. Computer programmers really are in the position of Lewis Carroll's Humpty Dumpty: when they use a word, it is them who chooses its meaning, and they can make it mean anything they like. The downside of this freedom is that it impedes communication, and accordingly – collaboration.

As a result, a single most valuable trait of a programming language – as a means of communication, as opposed to development of particular applications – is its simplicity. A good programming language should be simple to learn, and the programs written in it should be simple to understand (as simple, one could say, as possible, but no simpler).

Of course, there is nothing in a programming language alone that can prevent a programmer from using it in an obscure way, and therefore in addition to the use of appropriate means of expression, programmers need to obey a certain discipline, or a set of conventions. Needless to say, the conventions themselves should be clear and simple to follow, in order to avoid their misinterpretations.

One could ask whether it would be a good idea to use some natural language, such as English, to express computer programs. The big advantage of this approach is that most people already know at least one natural language, their so-called mother tongue.

However, there are some serious drawbacks of that idea. Natural languages tend to be vague, verbose, imprecise and structurally ambiguous[1]. People realized this fact long before the computer has been conceived, and this realization was reflected in the development of the language of mathematics, which has complementary advantages to those of natural languages: it is usually brief, non-ambiguous and precise[2].

Mark Turner and Gilles Fauconnier, the authors of an important cognitive science book "The Way We Think", noted that

> The development of formal systems to leverage human invention and insight has been a painful, centuries-long process. [...] In the twelfth century, the Hindu mathematician Bhaskara said, "The root of the root of the quotient of the greater irrational divided by the lesser one being increased by one; the sum being squared and multiplied by the smaller irrational quantity is the sum of the two surd roots." This we would now express in the form of an equation, using the much more systematically manageable set of formal symbols shown below. This equation by itself looks no less opaque than Bhaskara's description, but the notation immediately connects it to a large system of such equations in

[1] The idea of using natural language to program computers has been criticized at length by Edsger Dijkstra, who concluded that "machines to be programmed in our native tongues —be it Dutch, English, American, French, German, or Swahili— are as damned difficult to make as they would be to use"[17]

[2] However, certain problems with the way the language of mathematics is customarily used to lay out physics have been pointed out Gerald Sussman, who proposed to use a programming language for that purpose instead[77].

ways that make it easy to manipulate.[22]

$$\sqrt{(\sqrt{\frac{n}{k}}+1)^2 k} = \sqrt{k} + \sqrt{n}$$

One could therefore ask whether it might be a better idea to program computers in the language of mathematics, and indeed some successful attempts were made in this regard.

However, if we look at how the language of mathematics is usually being communicated, we find that it is rarely self-contained – that the concise mathematical formulas are often interleaved with more lengthy explanations provided in the form of natural language sentences, whether in text or near the blackboard. This "classroom" or "handbook" setup provides an explanation for the mathematicians' inclination to use short symbols that are easy to write quickly, although they do not, by themselves, provide any clues with regard to what they mean or how they should be pronounced.

There was an attempt to transplant this "handbook" approach to the domain of Computer Science called "Literate Programming", undertaken by the prominent mathematician Donald Knuth[53]. However, the industry practices show that this approach failed to become widespread.

A likely reason for this state of affairs is that it requires to maintain the correspondence between the program and its explanation, and there are no tools that could enforce this correspondence.

1.1.3 Referential transparency

A computer program, when perceived as a speech act, seems to be liberated from some of the constraints of both natural language and the language of mathematics, while imposing some of its own constraints.

Computer programs are usually written on computers, which aid not only in editing and displaying them, but also in paraphrasing them (which, in the parlance of programmers, is called *refactoring*[29]) and proving them correct.

If we look at industry practices, we'll notice, that there are also some things that the natural languages have in common with the so-called "industry best practices": just as there is a great deal of redundancy in natural languages, that minimizes the risk of misinterpreting an utterance, programmers are recommended to augment their programs with redundant features such as tests, assertions and type signatures[23] [29] [46] [59].

Another feature that is shared by natural languages and mathematics, but has only recently been gaining popularity among computer programmers, is called *referential transparency*, and is closely related to *the principle of compositionality.*

Referential transparency is a trait of languages' expressions that allows to substitute a reference to an object with another reference to the same

object (or – if possible – with that object itself), yielding a *co-extensional* expression. For example, in the expression

$$2 + 2 = 2 * 2$$

we can replace both left- and right-hand side of the = sign with some other expressions that has the same value, for example

$$2 + 2 = 4. \tag{1.1}$$

Of course, this rule applies not only to mathematics, but to ethnic languages as well. Incidentally, we can replace the name *William Shakespeare* with the phrase *the author of Hamlet* in an utterance, usually preserving its overall meaning.

Referential transparency usually applies to most expressions of our everyday language. However, there are two situations where it fails: context-sensitive terms (such as the words *this* or *you*) and so-called *intensional contexts*[30] (for example, we cannot transform the sentence *Lois Lane knows that Clark Kent is Clark Kent* into *Lois Lane knows that Clark Kent is Superman*, despite the fact, that both *Clark Kent* and *Superman* refer to the same entity).

The programming style that employs referential transparency is called *functional programming*, and it allows to comprehend the code in terms of the *substitution model of computation*[1].

For example, the following Python code that computes the *factorial* function in the so-called *imperative style* contains identifiers that are not referentially transparent:

```
def factorial(n):
    accumulator = 1
    while n > 0:
        accumulator = accumulator * n
        n = n - 1
    return accumulator
```

In every iteration, the meanings of the names `accumulator` and `n` are different: although they are only used in a single lexical context, there are many different invisible "execution contexts", in which those names are used. This phenomenon seems to have no counterpart neither in natural languages nor in the language of mathematics, where the meanings of words are either fixed, or depend only on some lexical context[3].

However, the same function can be defined in a referentially transparent manner with the use of recursion:

[3]The author remembers the following riddle from his childhood. A mean kid asks another kid (a victim): "I am me, you are you, which of us is the stupid one?". If the victim responds "you", then the mean kid claims that – according to what they settled earlier – the identifier "you" refers to the victim, so the victim admitted to be the stupid

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

Since no variable assignment appears in the code, we can evaluate, say, the expression `factorial(5)` by substituting the references to function at given points with their values at that point:

```
>>> factorial(5)
=== 5 * factorial(4)
=== 5 * 4 * factorial(3)
...
=== 5 * 4 * 3 * 2 * 1 * 1
=== 120
```

1.1.4 Historical perspective – data structures

The advantage of functional programs is that they tend to be more modular and reusable than their imperative counterparts. From the point of view of software engineering, they allow only a single way of passing information, namely – either by providing it as arguments to functions, or collecting it from functions' results. The lack of assignment doesn't allow to pass information through global variables, for instance, which results in cleaner systems that are easier to maintain. Also, if programs are written with pure functions, the order of their evaluation doesn't matter (due to the Church-Rosser theorem[41] [27]), which makes it easier to perform evaluations in parallel on different machines or CPU cores[3] [18].

However, this simplicity comes with a price. The users of imperative languages can tailor their programs to make the best use of the hardware that's going to be used to run it. Moreover, there is a considerable collection of solutions readily available for imperative programming languages, created under the assumption that *programs = algorithms + data structures*[82], [14].

The designers of programming languages quickly realized, that the *programs = algorithms + data structures* paradigm imposes an enormous cognitive strain on programmers. If they wish to use the advantages of a few particular data structures, they usually need to write code that blends some catalogued data structures together.

---

one. On the other hand, if the victim responds "me", the mean kid is also laughing, because he made the victim admit to be stupid. Note that in this case, the identifiers "me" and "you" were not modified. They preserved their original meanings, but they were shadowed by some new meanings, that referred to the old meanings.

One attempt to solve this problem was to separate data structure implementations from their interfaces. This idea was embodied in the *Standard Template Library* of the C++ programming language, which provided a few different data structures that could be used with a fixed set of generic functions[75] [74].

It was also at the core of the SQL family of languages, which actually were designed as a unified interface to a plenitude of various data structures.

Another idea was to provide a few basic, most commonly used data structures in the core language. Those typical data structures usually are: an ordered sequence of elements of arbitrary length (lists), an ordered sequence of elements of fixed length (tuples), a mapping between some key values and corresponding target values (dictionaries, usually implemented using hash tables), and unordered set of elements of arbitrary size (sets).

The latter approach has been employed by many popular programming languages, such as Perl, Python, PHP, Ruby or JavaScript. Although the performance of their versatile built-in data structures is usually worse than in the case of tailored solutions, it is often sufficiently good for practical applications, and the greater conceptual simplicity of the source code makes its development and maintenance easier.

An extreme version of this approach was proposed in 1958 by John McCarthy, who designed the LISP programming language[61]. LISP used one data structure to represent collections, namely – singly linked lists. They turned out to be expressive enough to embrace not only sets and multisets, but also dictionaries (as lists of key-value pairs).

Since its inception, singly linked lists became a predominant data structure in functional programming[4], which – in conjunction with the technique of garbage collection (also pioneered by McCarthy) allowed to develop programs devoid of state mutation.

Unfortunately, singly linked lists have some undesired properties that disqualify them in a number of applications. For example, the access time to a random element is linear (as opposed to constant, as it is the case for arrays), similarly to calculating the length of a list. Furthermore, modern CPUs are optimized to process data that is organized in vectors, and the non-locality of link reference operation may contribute to an increased number of cache misses, left alone the doubled memory consumption caused by the need to store additional pointer along with each element of the list.

For a long time, those deficiencies prevented functional programming from becoming widespread. Even the programming languages that were advertised as functional usually incorporated arrays in the repertoire of their primitive data structures, and specified their order of evaluation[66] (or provided some sophisticated means to do so[81]) in order to be able to use them in a predictive way.

In 2002, Phil Bagwell proposed a data structure called VList, which merged the functionality of linked lists with vectors. The interface to the

data structure remained unchanged compared to McCarthy's version, so the new data structure could easily replace linked lists in both compiled and interpreted code[4].

For exactly the same reason, it didn't provide the improvement one could hope for – functions that allocate resources in run time, without any help from static analysis, must remain ignorant to the way in which those allocated resources are used.

1.2 Formulation

Let's consider the following program – the variant of Quick-sort, written in the Haskell programming language:

```
1 qsort [] = []
2 qsort (p:xs) = (qsort below) ++ [p] ++ (qsort above)
3     where
4         below = filter (< p) xs
5         above = filter (>= p) xs
```

The central idea of the algorithm is expressed in the line 2: that the resulting sequence consists of the (sorted) elements smaller than pivot, followed by pivot, and then followed by the (sorted) elements greater than pivot.

Let's contrast it with the imperative version from [14]:

```
QUICKSORT(A, p, r)
1  if p < r
2       q ← PARTITION(A, p, r)
3       QUICKSORT(A, p, q − 1)
4       QUICKSORT(A, q + 1, r)
```

where PARTITION is defined as

```
PARTITION(A, p, r)
1  x ← A[r]
2  i ← p − 1
3  for j ← p to r − 1
4       if A[j] ≤ x
5            i ← i + 1
6            exchange A[i] ↔ A[j]
7  exchange A[i + 1] ↔ A[r]
8  return i + 1
```

These definitions come with a remark, that "to sort an entire array A, the initial call is QUICKSORT$(A, 1, length[A])$".

There are a few things to note here. First, that the imperative definition is much more difficult to follow, because array indexing adds another layer of indirection. Also, the imperative version is less composable, because it modifies its argument A, so if the old array ought to be used in some other part of the program, the programmer has to remember to make a copy.

On the other hand, the `qsort` function defined in Haskell can only superficially be called *quick*: both the `filter` function and the list concatenation operator `++` allocate new storage, and the time complexity of the `++` operator is proportional to the length of its leftmost argument. Furthermore, each use of the `filter` function traverses the input list once, so it may be traversed twice per each invocation of `qsort`. By contrast, the PARTITION function traverses the input array only once.

There is also a significant difference in the content of the data structures at the intermediate steps of computation: the `above` and `below` lists will contain elements in the order in which they appear in the `xs` list, while the result of PARTITION is generally difficult to determine.

The reason why this question doesn't matter too much is that the elements eventually end up sorted, no matter what both the initial and intermediate arrangements of elements are.

Clearly, there is a big difference between the functional `qsort` and the imperative QUICKSORT, because the latter is based on a clever idea of PARTITIONing an array (in place).

However, we can perceive the Haskell program as a sort of a section through its imperative counterpart. In particular, it should be possible to mechanically transform certain classes of functional programs so that they would use arrays instead of lists, and reuse previously allocated storage instead of allocating new one. This is what this work is going to be about.

1.3 Structure of this work

This work is organized in two parts – the purpose of the first part is to introduce the basic notions that are going to be needed to formulate the problem in a bit more rigorous terms. In particular, chapter 2 describes the source programming language that we wish to use for expressing our programs, and chapter 3 proposes a simple register machine model and its instruction set, which is the target of our transformations/optimizations. In chapter 4, we present a general transformation which allows to execute programs in our source language on the target machine (this transformation is traditionally called *compilation*). Chapter 5 gives an overview of a system for reasoning about programs expressed in (a subset of) the source language.

In the second part, we develop some techniques for optimizing various

classes of programs. In chapter 6, we develop a technique that allows to transform various basic functions that operate on lists into procedures operating on arrays. Chapter 7 deals mostly with Quicksort, and although it fails to deliver working solutions, we hope that it at least arrives at some valuable conclusions.

In every chapter, there is a lot of code illustrating the ideas being presented or elaborated. The code is the integral part of the work, and we encourage the reader to actually read it (rather than skip it). Except for the chapter 5, all the code was actually tested (and therefore we suspect that programs in chapter 5 almost certainly contain some bugs).

All the code is written using the Scheme programming language. We've heard that some people find it difficult to comprehend the programs written in that language because of the multitude of parentheses that can be used in some more complex expressions. In our hope that the hesitant readers overcome their repulsion, we assure that the code is indented in a way that was supposed to facilitate the comprehension of the programs, rather than confuse the readers.

We admit that – although the syntax of the Lisp family of programming languages turned out to be perfect for our task (and we challenge the readers to prove us wrong) – the typesetting of Lisp programs clearly requires some elaboration.

On the other hand, we have to say that the habit of building and testing programs bit by bit gave us a lot of confidence in the quality of our work. In particular, this approach allowed us to find out that there are some problems with the functional implementation of Quicksort proposed in [45] (cf. appendix D)

1.4 Related work

Our work partially overlaps on some efforts that were undertaken in order to reduce the burden of garbage collection to compile time ([5], [13], [40], [42], [45]) and of interfacing arrays in functional languages ([6], [81]).

The idea of using a theorem-proving system for optimizing programs appears in [35], and is – to an extent – explored in [12].

An alternative approach to the problem of tackling data structures in functional program – not by transforming the programs, but by designing the data structures in such a way that makes them more suitable in immutable setup – can be found in [64] and [4] (among others).

1.5 Acknowledgements

First and foremost, I would like to thank my family and my girlfriend for supporting my crazy decision of returning to the University to fill the gap

in my education. In particular, I appreciate the patience of Dorota, that I have been permanently overusing.

I am grateful to my dearest friend Ścisław Dercz vel Michał Stańczyk, for countless stimulating discussions and his bright critical remarks regarding this work (sorry for giving out your secret identity, you uneducated ragtag).

I am also grateful to the teachers that I've had a pleasure to work with during this rather lengthy period of my education, in particular to Wiesław Pawłowski for sharing his fascination in logic, Janusz Dybizbański and Maciej Dziemiańczuk for all the fun I've had during their classes, Tomasz Dzido for giving me an opportunity to spread the ideas of functional programming in the realm of combinatorics, Paweł Żyliński for appreciating the succinctness of the solutions that I have been bringing to him, Grzegorz Madejski for encouraging me to work on [39], that in a way was a prelude to this work, and of course my supervisor Christoph Schwarzweller who provided me with the perfect conditions for working on this thesis. I also deeply appreciate the wise advice of Andrzej Szepietowski who pointed me to him.

I owe a lot to Paulina Śliwka and Alicja Zurita-Kwapińska from the Dean's office, who showed me so much kindness and support, and Andrzej Borzyszkowski, whom I have been troubling constantly.

The design of the virtual machine from chapter 3 was directly inspired by the course in the Theory of Computation that I took during the year that I've spent studying Philosophy at the University of Warsaw. I would like to thank Marcin Mostowski for being a demanding teacher, and for assigning me the task of preparing a note about the RAM machines [38] (although the note is in Polish, I was nicely surprised to have discovered that the same concept was described in a popular book about philosophy [16]).

I also wanted to thank Sophia Gold for sacrificing her time to read this work, and for pointing me to a new whole realm of papers on the related subjects (I have to admit that I am extremely impressed with you as a person), and Carl Eastlund for helping me out with inductive proofs, and for co-authoring "The Little Prover".

To be honest, I could go on and on with the list of people to whom I am grateful, and whose existence contributed to this work. I don't think that I would manage to bring this work into existence without the three years that I've spent at the University of Gdańsk studying Philosophy, being influenced by a lot of great teachers, including Stanisław Judycki, Martyna Koszkało, Rafał Urbaniak, Andrzej Leszczyński, Aleksandra Pawliszyn, Iwona Krupecka, Wojciech Bęben and many others, as well as my inspiring and enthusiastic friends and colleagues. It was the best time in my life.

I may have failed to appreciate the value of the time that I've spent studying Automatic Control and Robotics at the Technical University of Gdańsk, where – despite the "industrial" hostility of that place – I have also met a couple of soul mates, notably Piotr Suchomski, who taught me to love Mathematics.

I should also mention Radek Potyraj and Andrzej Macuk, who created the right conditions at the Fellows company, which allowed me to try to continue the studies. I am grateful to Michał Janke for dragging me to their company, and I regret that in the end the things didn't turn out as good as they could have. I appreciate the attitudes of my colleagues at work, who have been showing interest in this work and my opinion and expertise, making me feel as someone important.

My apologies to everyone who might have felt that his or her name is missing here. For purely ecological reasons, I'm unable to do the justice to all of you here. (This also includes a lot of people that I never had any opportunity to meet in person, and whose work I had to admire from the distance, be it spatial or temporal.)

Part I

The Organon

2

The source language

First we must define the terms "noun" and "verb", then the terms "denial" and "affirmation", then "proposition" and "sentence".
– Aristotle, *On Interpretation*

2.1 Overview

In the previous chapter, we presented two variants of seemingly the same algorithm – one written in the functional language Haskell, and the other presented in its classical imperative form.

In this work, we are going to analyze the general conditions that need to be satisfied in order to be able to transform functional programs that operate on lists into equivalent imperative programs that operate on arrays, and search for the means that can be used in order to detect whether those conditions are actually satisfied.

We shall begin with defining a programming language that will be used to express algorithms on data structures that are supposed to be optimized, and specifying a computational model that will be the target of our transformations.

An ideal programming language for our purpose would possess the following traits: it would be simple to describe, simple to process and powerful enough to express any computable function.

For those reasons, we decided to choose a purely functional subset of the Scheme programming language[66], devoid of the `set!` instruction and the `call-with-current-continuation` control operator. Furthermore, although the specification of Scheme defines the strict (or applicative) order of evaluation, the programs presented in this work shall never rely on this. Lastly, as the point of this work is to describe techniques for optimizing programs that operate on lists, our subset of Scheme need not support vectors, in spite of their presence in the specification. Likewise, we shall ignore the

support of character strings in the language, as well as its numerical tower (for our purpose, the support for integer numbers should be sufficient).

There's plenty of excellent education resources available for the Scheme programming language [1] [28] [34] [31] [33] [20] [69] [39], so only a brief introduction to the language will be presented here.

As Scheme is not only going to be the source language, but also – occasionally – the intermediate language for compilation, we are also going to describe some imperative features that are going to appear in the generated code.

2.2 Syntax

Syntactically, Scheme employs a fully parenthesized prefix notation built around the so called s-expressions that could be described with the following BNF-style grammar:

```
<s-expression> ::= <atom> | (<s-expression> . <s-expression>);

<atom> ::= () | <symbol> | <number> | #true | #false;
```

A `<number>` is a sequence of decimal digits. A `<symbol>` is a sequence of non-whitespace and non-parenthetic characters that does not begin with `#` and cannot be interpreted as a `<number>`.

The tokens are separated either by white spaces or by parentheses.

In addition, the *pair notation*, e.g. `(a . (b . (c . d)))` is equivalent to `(a b c . d)`, and in particular `(a . (b . (c . ())))` (the *last tail* being an empty list) is equivalent to `(a b c)`.

The expressions `(quote` *x*`)`, `(quasiquote` *x*`)`, `(unquote` *x*`)` and `(unquote-splicing` *x*`)` can be abbreviated as `'`*x*, `` `x ``, `,`*x* and `,@`*x*, respectively.

The semicolon character (`;`) is used to mark comments that span to the end of line, or – if the semicolon is directly preceded by the `#` character – until the end of the following s-expression.

2.3 Semantics

The value of a number is that number itself (thus we say that numbers are self-evaluating). The value of a symbol is the value that has been bound to that symbol. A symbol is said to be bound if either:

- it is predefined by the language,
- it has been defined using a `define` special form (or its derivative) in the current lexical context,

- it appears in the argument list of a surrounding `lambda` expression (or its derivative).

If a symbol is unbound, the meaning of a program is undefined.

An s-expression of the form `(operator operands ...)` is called a *combination*. A combination can either be a special form, or a function application.

2.3.1 Special forms

Special forms can either be primitive or derivative.

2.3.2 The primitive special forms

The subset of Scheme of our interest contains the following primitive special forms: `lambda`, `define`, `if` and `quote`.

The `lambda` form

The `lambda` form is used to create function, and consists of two parts: a *list of arguments* and a *body*. For example, a function of two arguments whose value is its first argument can be created in the following way:

$$\texttt{(lambda}\ \underbrace{\texttt{(x y)}}_{arguments}\ \underbrace{\texttt{x}}_{body}\ \texttt{)}$$

Note that the symbols that appear in the argument list must be unique, i.e. it must not be the case that the same symbol appears on the list more than once. Also, the body must be a valid Scheme program.

The list of arguments need not be proper. In such cases, the dotted tail argument represents a list of optional (variadic) arguments.

Although the `lambda` form is sufficient to express any computation [41], including operations on natural numbers as well as lists[1], Scheme provides some additional primitive forms for convenience.

The `define` form

The `define` form is used to extend the current lexical context with a new binding. It takes two operands: a symbol that shall be bound with a value (*definiendum*), and an expression whose value shall be bound to the symbol (*definiens*). For example, the definition

$$\texttt{(define}\ \underbrace{\texttt{x}}_{definiendum}\ \underbrace{\texttt{5}}_{definiens}\ \texttt{)}$$

causes `x` to be bound to the value `5`. The `define` special form is the only primitive special form that is not considered an expression.

Also, since – unlike `lambda` – the `define` form does not create a new scope, but rather extends the current one, it allows to define functions that are recursive.

Although the basic form of `define` is `(define name value)`, we are going to treat usages like `(define (f x) ...)` as short-hand for `(define f (lambda (x) ...))` and so on[1].

The `if` form

The `if` expressions take the form

```
(if <condition> <consequent> <alternative>)
```

If the `<condition>` evaluates to `#false`, then the value of the whole expression is value of `<alternative>`; otherwise, it is the value of `<consequent>` (therefore, every value other than `#false` is considered to be true in the context of an `if` expression. Such values will be referred to as *truth-ish* throughout this text).

The `quote` form

The `quote` form is used to input literal data. For example, the value of the expression `(quote x)` is the symbol `x`. The `quote` operator is redundant for self-evaluating expressions (e.g. numbers), so there is no practical difference between expressions `(quote 1)` and `1`.

The operator is not idempotent, though: the value of the expression `(quote (quote 1))` is a list of two elements, whose first element is the symbol `quote`, and whose second element is the number `1`.

As it was noted in the section describing the syntax, the expression `(quote x)` can be abbreviated as `'x`, so – consequently – the expression `(quote (quote 1))` can equivalently be written as `''1`, and also `'(quote 1)` and `(quote '1)`.

The `begin` and `set!` forms

Although the source programs that we are going to write are purely applicative, the resulting program will occasionally contain some procedural constructs. The `begin` form is used for sequencing operations. Its value is the value of the last expression in that form (in particular, `(begin x)` and `x` are equivalent).

[1] In particular, the `(define ((f x) y) ...)` can be treated as a short-hand for `(define f (lambda (x) (lambda (y) ...)))`. This generalized feature, called *curried definitions*, is not provided by most implementations of Scheme.

The `set!` form is used for performing assignment. For example, the value of the expression[2]

```
(begin
  (define x 5)
  (set! x (+ x 1))
  (* x x))
```

is the number 36. There is no need to use `begin` in the body of the `lambda` form: `(lambda args (begin actions ...))` and `(lambda args actions ...)` are equivalent. Most typically, the `begin` form is used in one or both branches of the `if` form.

The derivative forms

Scheme offers a mechanism that allows to define derivative special forms, often refereed to as syntax extensions or macros. It also predefines a set of helpful derivative special forms: `let`, `let*`, `and`, `or`, `cond` and `quasiquote`.

The `let` form

The `let` form is used to create local bindings, and is defined so that

```
(let ((<variable1> <value1>)
      (<variable2> <value2>)
      ...)
  body)
```

expands to

```
((lambda (<variable1> <variable2> ...) body) <value1> <value2> ...)
```

The `let*` form

In the case of the `let` form, the bindings of variables to all values occur simultaneously, so for example in the expression

```
1 (let ((x 2)
2       (y (+ x 1)))
3   (+ x y))
```

[2] It may be questionable whether that program can actually be called an expression, as it introduces a definition into its current scope. The snippet presents a very bad style of programming, but it also presents the meaning of the special forms discussed in this section.

the symbol `x` in the line `2` does not refer to the value from the binding in the line `1`, but to the value from the outer scope of the expression. This behavior is often undesired in practice.

For that reason, Scheme provides a sequential variant of `let` called `let*`, which is defined so that

```
(let* ((<variable1> <value1>)
       (<variable2> <value2>)
       ...)
  <body>)
```

expands to

```
(let ((<variable1> <value1>))
  (let* ((<variable2> <value2>)
         ...)
    <body>))
```

The `and` form

The `and` form expresses logical conjunction that uses short-circuited evaluation [61] [71], defined so that

```
(and <condition1> <condition2> ...)
```

expands to

```
(if <condition1>
    (and <condition2> ...)
    #false)
```

and

```
(and <final-condition>)
```

expands to

```
<final-condition>
```

thereby causing the `and` clause to expand either to `#false` or to the value of its `<final-condition>`.

The `or` form

Like the `and` form, the `or` special form performs evaluation in a short-circuited manner, and is also defined to evaluate to the value of its succeeding clause. This last requirement causes a slight complication under the strict model of evaluation, as it requires to capture the value of expression in order to make sure that it is evaluated only once:

```
(or <condition1> <condition2> ...)
```

expands to

```
(let ((##result <condition1>))
  (if ##result ##result (or <condition2> ...)))
```

where `##result` is a unique identifier that does not appear anywhere in the rest of the code. As in the case of `and`, the expression `(or <final-condition>)` simply expands to `<final-condition>`.

The `cond` form

The `cond` form is used to avoid nested `if`s[3]. The expression

```
(cond (<condition1>
       <value1>)
      (<condition2>
       <value2>)
       ...)
```

gets expanded to

```
(if <condition1>
    <value1>
    (cond (<condition2>
           <value2>)
          ...))
```

The `when` and `unless` forms

In the generated imperative code, we may sometimes want to use the forms `when` and `unless`. The former is defined so that

```
(when condition actions ...)
```

expands to

```
(if condition (begin actions ...))
```

and the latter – so that

```
(unless condition actions ...)
```

expands to

```
(if (not condition) (begin actions ...))
```

[3] Actually, the `cond` form present in the Scheme language is a bit more complex than presented here.

The `quasiquote` form

The `quasiquote` form (in conjunction with two helper keywords, namely – `unquote` and `unquote-splicing`) is used for creating data conveniently. The convenience stems from the fact, that the syntaxes `(quasiquote x)`, `(unquote x)` and `(unquote-splicing x)` can be abbreviated as `‘x`, `,x` and `,@x`, respectively. This allows to create data that contains some variable elements in it, for example the value of the expression

```
(let ((x 3)
      (y '(5 6 7)))
  `(1 2 ,x 4 ,@y 8))
```

is the list `(1 2 3 4 5 6 7 8)`.

Non-standard derivative forms

Receiving multiple values

Scheme allows functions to return more than one value using the `values` form. For example, the form `(values 1 2 3)` returns three values: `1`, `2` and `3`.

Normally (e.g. in the context of a function call) only the first value is taken into account, and the remaining ones are ignored. Scheme provides a special form `call-with-values`, which allows to capture the remaining return values. For example, `(call-with-values (lambda () (values 1 2 3)) list)` passes the values `1`, `2` and `3` as subsequent argument to the `list` function (effectively creating the list `(1 2 3)`).

We are going to assume here, that the `let*` form can be abused to receive multiple values[21], so that, for example

```
(let* ((a b c (values 1 2 3)))
  (list a b c))
```

is equivalent to

```
(call-with-values
  (lambda () (values 1 2 3))
  (lambda (a b c) (list a b c)))
```

Note that the multiple return value feature will never be used in this work when Scheme is to be treated as the subject language (i.e. as data to be processed by compilers, interpreters, theorem provers and other transformers).

The `is` form

We shall also be using a non-standard extension to the Scheme language, proposed by the author of this work in [37]. In short, it introduces the `is` special form, such that

```
(is a related-to? b)
```

expands to

```
(related-to? a b)
```

In addition, it treats the `_` (underscore) symbol in the argument position specially. For example,

```
(is _ related-to? x)
```

expands to

```
(lambda (_) (related-to? _ x))
```

2.3.3 Function applications

If a combination is not a special form, it is treated as a function application. A function that is applied to can either be a primitive function, or a function created with a `lambda` form.

Functions created with a `lambda` form

As noted before, the `lambda` form contains a list of arguments and a nested Scheme expression, referred to as the body of the `lambda` form.

The value of the application of a function created by a `lambda` form is simply the value obtained by evaluating its body in the lexical scope extended with bindings of its arguments to the values of the operands of the combination.

For example, the value of the expression

```
((lambda (x y) x) 5 10)
```

is the value of the expression `x` in the environment where the symbol `x` is bound to `5` and the symbol `y` is bound to `10`.

2.3.4 A note on lexical scoping

Scheme is said to be a *lexically scoped* language, as opposed to *dynamically scoped*. *Lexical scope* means that a symbol which occurs free in the body of a `lambda` expression (i.e. it does not appear on the argument list of the `lambda` expression) refers to the meaning from the context of the definition of that `lambda` expression, rather than the context of its usage (as it is the case for dynamic scope).

For example, if we

```
(define square (lambda (x) (* x x)))
```

it is apparent that the symbol `*` occurs free in the definiens. It is typically bound to a procedure that multiplies its arguments (see *Primitive functions* section below).

The question is, what would be the value of the expression

```
(let ((* +))
  (square 5))
```

where the symbol `+` is bound to a procedure that adds its arguments.

In a dynamically scoped language, the value of that expression is `10`, whereas in a lexically scoped language it is `25`.

2.3.5 A note on recursive definitions

Note that there is a significant difference between the bindings created using the `define` form and the ones obtained introduced by `let` or `lambda`. In particular, `lambda` creates a new scope, whereas `define` introduces new bindings to the current scope.

This difference becomes particularly significant if one attempts to define a recursive function. Because of that, the following expression of the *factorial* won't work as one could hope:

```
(let ((! (lambda (n)
           (if (= n 0)
             1
             (* n (! (- n 1)))))))
  (! 5))
```

The `!` symbol in the body of the `lambda` expression refers to the binding in the environment in which the `lambda` expression is evaluated (the so-called *top-level* environment), and not the environment introduced by the `let` form (in which the `!` symbol gets bound to the value of the `lambda` expression).

This is not the case with the `define` forms. The `define` form introduces a new binding to the current environment, so the definition

```
(define ! (lambda (n)
            (if (= n 0)
              1
              (* n (! (- n 1))))))
```

actually allows to compute a factorial. It can be noted however[41], that the `lambda` form is sufficient not only for naming intermediate values of computation, but also for expressing recursive functions, through the application of the so-called *fixed-point combinators*, i.e. a family of operators ∇, such that for any y, $\nabla\, y = y\, \nabla\, y$.

An example of a fixed-point combinator is called `Y`-combinator, and is defined as

```
(define Y (lambda (f)
            (let ((R (lambda (x) (f (x x)))))
              (R R))))
```

Using this combinator, we can express the recursive factorial function without the use of the `define` form (i.e. using only derivatives of the `lambda` form) in the following way:

```
(let* ((F (lambda (f)
            (lambda (n)
              (if (= n 0)
                1
                (* n (f (- n 1)))))))
       (! (Y F)))
  (! 5))
```

Such interpretation should work fine in lazy or call-by-name languages (such as Haskell or Lazy Racket), but will fail in languages with strict semantics (such as Scheme or Python or JavaScript or Java), that evaluate their arguments prior to function application, because the `Y` combinator would expand *ad infinitum*[4].

Therefore we need a fixed point combinator that yields a *thunk* (i.e. a function of no arguments), and that this thunk is evaluated before the recursive call, so that the expanded version of our factorial would look like this

[4] Note that it is typical for strict languages to support assignment, which allows to express recursion rather easily, e.g.:

```
(let ((! #false))
  (set! ! (lambda (n)
            (if (= n 0)
              1
              (* n (! (- n 1))))))
  (! 5))
```

Since the `lambda` expression is created in the context where the variable `!` is already defined (as the value `#false`), there is no problem with it referring to itself.

(note the additional pair of parentheses around the nested `f` and generating `(Z F)`):

```
(let* ((F (lambda (f)
            (lambda (n)
              (if (= n 0)
                1
                (* n ((f) (- n 1)))))))
       (! ((Z F))))
  (! 5))
```

where the fixed point combinator `Z` is defined (non-recursively) as

```
(define Z (lambda (f)
            (let ((R (lambda (x)
                       (lambda () (f (x x))))))
              (R R))))
```

While this construction's purpose is to implement recursion in strict languages, it should work in lazy languages as well.

2.3.6 Primitive functions

The considered subset of the Scheme programming language also contains a handful of primitive functions that operate on the primitive data types, i.e. numbers, lists and symbols.

Primitive functions that operate on numbers

The set of primitive functions that operate on numbers shouldn't be surprising: it consists of functions such as addition (`+`), multiplication (`*`), division (`/`) and subtraction (`-`). The functions `*` and `+` accept arbitrary number or arguments (for zero arguments, they evaluate to the neutral element of multiplication and addition, respectively).

The functions `/` and `-` can take one or more arguments. The combinations `(/ x)` and `(- x)` are equivalent to `(/ 1 x)` and `(- 0 x)`, which evaluate to the inverted and negated value of `x`, respectively.

The values of the expressions `(-` $\mathtt{a}_1$ $\mathtt{a}_2$ `...` $\mathtt{a}_n$`)` and `(/` $\mathtt{b}_1$ $\mathtt{b}_2$ `...` $\mathtt{b}_k$`)` (where $\mathtt{b}_i \neq 0$ for $1 < i \leq k$) are the numbers $((...(\mathtt{a}_1 - \mathtt{a}_2) - ...) - \mathtt{a}_n)$ and $((...(\mathtt{b}_1/\mathtt{b}_2)/...)/\mathtt{b}_k)$.

In addition, there are binary functions `quotient` and `remainder` whose names are rather self-explanatory.

The unary predicate `number?` can be used to check whether a given object is a number.

There is also the `random` function that takes an integer `n` and evaluates to a random integer between `0` and `n-1`. Note that although programs that contain a call to the `random` function are no longer referentially transparent, they can still be analyzed in terms of the substitution model of computation.

Primitive functions that operate on lists

The basic function for constructing lists is called `cons`. It takes two arguments. The value of the expression `(cons a b)` is a (typically) newly allocated pair `(a . b)` (also called a *cons cell*) from the garbage-collected heap.

In order to retrieve the values from a cons cell, one can use the accessor functions `car` and `cdr`. In particular, for any values a and b, the expression `(car (cons a b))` evaluates to a and the expression `(cdr (cons a b))` evaluates to b.

The predicate `pair?` can be used to check whether a given object is a cons cell, so for any values a and b, the expression `(pair? (cons a b))` evaluates to truth-ish value. On the other hand, for any value x, if x is an atom, then `(pair? x)` evaluates to `#false`.

The `apply` function can be used to apply a function to a list of arguments. In particular, if `l` is a list `(`a_1 a_2 `...` a_n`)`, then `(apply f l)` is equivalent to `(f` a_1 a_2 `...` a_n`)`.

Primitive comparison predicates

The notion of identity in Scheme may seem a bit complex at first. The primitive predicate `eq?` can be used to check whether two expressions evaluate to the same object, that is, an object that occupies the same space in the computer memory.

In particular, it need not be the case that `(eq? (cons 1 2) (cons 1 2))`, because the evaluation of each expression `(cons 1 2)` may allocate new memory, instead of re-using the already allocated.

On the other hand, it is guaranteed that two instances of a symbol with the same shape are `eq?`, for example `(eq? 'abc 'abc)` evaluates to a truth-ish value, and that two symbols with different shapes are not `eq?`, so for example `(eq? 'abc 'def)` evaluates to `#false`. Also, it is guaranteed that the empty list, i.e. `'()`, is always `eq?` to itself.

Situation gets more complicated in the case of numbers. Typically, two instances of the same number can be `eq?` if the numbers are small enough to fit into machine word. However, since Scheme supports arbitrary precision arithmetic, additional storage may be allocated on the heap to store results of arithmetic operations. In such cases, two instances of the same number may not be `eq?`.

This is why Scheme provides a separate predicate, `=`, that is used for checking numerical equality. In addition, it provides predicates `<`, `<=`, `>` and `>=` to check whether their arguments form ascending, non-descending, descending and non-ascending sequences, respectively.

2.3.7 Non-primitive functions defined in the language

In addition to the primitive functions, Scheme comes with a set of functions that are predefined, although they can be easily defined in terms of the primitives.

The notion of identity

The notion of identity based on memory address is very difficult to reason about, and its applicability is very limited. There is a much more natural and deterministic notion of identity that can be defined in terms of the presented primitive functions:

```
(define (equal? a b)
  (or (eq? a b)
      (and (pair? a)
           (pair? b)
           (equal? (car a) (car b))
           (equal? (cdr a) (cdr b)))
      (and (number? a)
           (number? b)
           (= a b))))
```

The above definition reads as follows. Two objects are `equal?` either if they are `eq?`, or if they are both `pair?`, their `car`s are `equal?` and `cdr`s are `equal?`, or they are both `number?` and they are `=` (numerically equal).

The actual `equal?` function available in Scheme is a bit more powerful, as it can take arbitrarily many arguments, and evaluates to `#false` if any two differ (in the sense implied by the above definition) from each other.

List processing

The `list` function evaluates to a list of its (evaluated) arguments, for example `(list 1 (* 1 2) (+ 1 (* 1 2)))` evaluates to the list `(1 2 3)`. The `list` function could be defined simply as `(lambda x x)`, because if the list of arguments of a `lambda` expression is improper, then they are captured in a list and bound with the dotted tail of the argument list.

The `map` function takes a n-ary function `f` (for $n \geq 1$) and n lists of length k and returns a new list of length k such that its elements are obtained from application of `f` to the subsequent n-tuples from the list. In other

words, given lists $L^{(1)}, L^{(2)}, ..., L^{(n)}$, where $L^{(i)} = (a_1^{(i)}\, a_2^{(i)} \ldots\, a_k^{(i)})$, `(map f` $L^{(1)}\, L^{(2)} \ldots\, L^{(n)}$`)` produces a list `((f` $a_1^{(1)} a_1^{(2)} \ldots\, a_1^{(n)}$`) (f` $a_2^{(1)} a_2^{(2)} \ldots\, a_2^{(n)}$`) ... (f` $a_k^{(1)} a_k^{(2)} \ldots\, a_k^{(n)}$`))`.

Of course, in the simplest (and most frequently used) case, i.e. when $n = 1$ and $L = (a_1\, a_2 \ldots\, a_k)$, the expression `(map f` L`)` evaluates to a list `((f` a_1`) (f` a_2`) ... (f` a_k`))`.

Another frequently used function is called `filter`. It takes a predicate `p?` and a list `L`, and returns a new list that contains only those elements from `L` that satisfy the predicate `p?`. The original order of elements is preserved. The expression `(filter p? L)` can therefore be read as "such elements of `L` that `p?`", for example, the expression `(filter (is _ > 5) L)` can be read as "such elements of `L` that are greater than `5`".

Non-standard functions

There are a few functions that are going to be used in this work that are not a part of the standard Scheme. They are explained here briefly, and their definitions are given in the appendix A.

Quantifiers

It is common in logic and mathematics to express statements using the quantifiers *for all* ($\forall$) and *exists* ($\exists$). For example, the sentence "All men are mortal" could be translated to predicate calculus as $\forall_x[man(x) \Rightarrow mortal(x)]$. We usually assume that there is a domain (universe) of all objects that we can talk about.

In Scheme, we usually have to be a bit more specific. It is customary to define two functions, usually called `every` and `any`, that take a predicate and a list of objects from the domain, and evaluate to `#false` if every object from the list satisfies the predicate (in case of `every`) or there is no object in the list that satisfies the predicate (in case of `and`), and otherwise evaluate to some truth-ish value.

So while the most faithful translation of the above example to Scheme would have the form

```
(every (lambda (x)
         (if (man? x)
           (mortal? x)
         ;else
           #true))
  THE-UNIVERSE)
```

it is much more customary to assume that we have a list of all men, and check whether each element of that list satisfies the predicate `mortal?`, i.e. `(every mortal? men)`.

Set operators

Lists can interpreted not only as sequences, but also as sets. On such occasions, the order of elements on the list becomes immaterial. An `element` is thought of as being a member of a set represented by a list if it is a `member` of that list, which is denoted as `(member element list)` or `(is element member list)`.

One can easily define the usual set operators such as `union`, `intersection` and `difference`. However, the order of elements contained in the result of these operations is undefined and should not be relied on (in particular, it might be the case that any of these functions called with the same arguments return the result in a different order upon each invocation).

Folding

Suppose that we are given a list, for example $[a_1, a_2, a_3, ...]$, and a binary operator $\circ$. The simplest variant of the $fold$ operation computes the value of the expression $(a_1 \circ a_2 \circ a_3 \circ ...)$.

Note that the above formulation is ambiguous. For example,

$$fold(\circ, [a_1, a_2, a_3, a_4])$$

i.e.

$$(a_1 \circ a_2 \circ a_3 \circ a_4)$$

can be interpreted as either

$$(((a_1 \circ a_2) \circ a_3) \circ a_4) \tag{2.1}$$

or

$$(a_1 \circ (a_2 \circ (a_3 \circ a_4))) \tag{2.2}$$

or

$$((a_1 \circ a_2) \circ (a_3 \circ a_4)). \tag{2.3}$$

The interpretation (2.1) is called *left fold* and the interpretation (2.2) is called *right fold*.

Of course, if the operator $\circ$ is associative, the interpretation is (by definition) insignificant from the denotational point of view (although the amount of resources used by those interpretations might differ under various circumstances).

Also, it is easy to see that, for the above $fold$ operation to make sense, it must be the case that $\circ : A \times A \mapsto A$.

This requirement can be loosened a bit, though. By introducing additional argument e, we allow the operator to have a type $B \times A \mapsto B$ in the case of the left fold, or $A \times B \mapsto B$ in the case of the right fold: then, *fold-left*$(\circ, e, [a_1, a_2, ..., a_n])$ is interpreted as $((...((e \circ a_1) \circ a_2) \circ ...) \circ a_n)$, and *fold-right*$(\circ, e, [a_1, a_2, ..., a_n])$ is interpreted as $(a_1 \circ (a_2 \circ (... \circ (a_n \circ e)...)))$.

The purpose for the presence the additional argument e can be explained as follows. Imagine that you have a state of the world, S, and a list L (possibly infinite) of actions that occur in the order as they appear on the list. There's an update function called $\triangleleft$ that takes a state and an action and returns an updated state. Under such circumstances, the evolution of the world can be modeled as *fold-left*$(\triangleleft, S, L)$.

If the operation $\circ$ has a neutral element (i.e. an element $\mathbb{1}$ such that, for any valid x, $\mathbb{1} \circ x = x$ in the case of left fold, or $x \circ \mathbb{1} = x$ in the case of right fold), it is often convenient to choose it as the argument e.

Pattern matching

We can write programs that operate on s-expressions using the primitive functions `pair?`, `car`, `cdr` and `eq?`. For example, if we want to know whether a given s-expression `exp` represents a sum of exactly two elements, say, `(+ a b)` (for some `a` and `b`), we can write a compound condition

```
(and (pair? exp)
     (eq? (car exp) '+)
     (pair? (cdr exp))
     (pair? (cdr (cdr exp)))
     (eq? (cdr (cdr (cdr exp))) '()))
```

If we would like to capture the first and second operand to plus, using, say, the `let` from, we would obtain:

```
(if (and (pair? exp)
         (eq? (car exp) '+)
         (pair? (cdr exp))
         (pair? (cdr (cdr exp)))
         (eq? (cdr (cdr (cdr exp))) '()))
    (let ((a (car (cdr exp)))
          (b (car (cdr (cdr exp)))))
       ;; here a and b are bound to the first
       ;; and the second operand of +
       ...)
    ;;else
    ...)
```

However, it is very difficult to analyze this sort of code (whose task is rather simple). It is therefore convenient to extend the syntax of Scheme with the `match` macro[5], so that the expression

[5] This document uses a subset of the syntax proposed in [83]. A portable implementation of this pattern language can be obtained from `http://synthcode.com/scheme/match.scm`.

```
(match exp
  (('+ a b)
   ;; here a and b are bound to the first
   ;; and the second operand of +
    ...)
  ;; possibly other matches go next
  ...)
```

expands to the above condition and binding list. Furthermore, we can establish a convention, that whenever a compound expression appears in the place of an argument in the `lambda` form (or any derivative form, such as `let` or `let*`), it gets pattern-matched, so for instance `(lambda ((a . b)) body)` would be interpreted as `(lambda (x) (match x ((a . b) body)))` (where the variable `x` does not occur free in the `body` form). The `_` (underscore) symbol has a special meaning: it matches anything, but does not get bound to any value.

We will sometimes be making use of a bit more exotic (and less obvious) feature of the pattern matcher, namely – the `...` (ellipsis) operator. It behaves similarly to the `,@` (`unquote-splicing`) operator in that it "unsplices" a list:

```
(match '(1 2 3 4 5)
  ((x y ... z)
   ;; x is bound to 1,
   ;; y -- to the list (2 3 4),
   ;; and z -- to 5
   ...))
```

2.4 The meta-circular evaluator

The subset of Scheme that we chose for the purpose of this work is powerful enough to express itself [6]

Since in this setup Scheme is used as both the language that we talk about and the language that we use to describe that language, this construct is called a *meta-circular evaluator*, as Scheme becomes a *metalanguage* for *itself*.

As a result, everything that has been said in this chapter about the semantics of Scheme, can be worded more precisely and rigorously using a formal notation.

In order to make the code clearer, we shall restrict the object language even more. First, we shall assume, that the `define` form does not appear in our program. Indeed, a mechanical procedure for transforming a program

[6] As it was noted by Christian Queinnec, "literature about Lisp rarely resists that narcissistic pleasure of describing Lisp in Lisp"[65], and here we make no exception.

containing the `define` forms into a program that only uses `lambda`, `let` and `let*` (and a fixed-point combinator) is given in the appendix B.

Also, for the sake of brevity, we shall ignore the issues concerning user-defined syntactic extensions. We shall also assume that the programs do not reuse either of the syntactical keywords (like `quote` or `lambda` or `if`), so for example we shall not be concerned with programs such as

```
((lambda (lambda)
    (lambda lambda))
 (lambda (quote)
    (quote quote)))
```

There is a general technique commonly referred to as *α-conversion*[73] for dealing with cases like this (where each of the variables bound with the `lambda` form or its derivative is renamed before the evaluation).

Lastly, we will assume that no derivative forms such as `let`, `let*`, `match`, `quasiquote`, `and` or `or` are used in the evaluated program, i.e. that all such forms have been expanded prior to the evaluation. The appendix C explains in detail how these derivative forms can be converted to primitive forms.

2.4.1 The core evaluator

The core of our evaluator is the `value` function, which produces the value of a given expression. It consists of a case analysis over the structure of expression. In particular, the ability to create new procedures using the `lambda` form plays the key role for the language's expressive power.

In order to manipulate the procedures, we need to come up with some way of representing them. Since the only types of objects that are available in our language are symbols, numbers, pairs and procedures, we'd need to form our representation from some combination of them[7].

We decided to represent procedures as tagged list, whose first element is a special symbol `procedure-tag`[8]:

```
(define (value expression environment)
  (match expression
    (('lambda args body)
     `(procedure-tag ,args ,body ,environment))
```

[7] There would be very little point in using procedures to represent procedures, as the main purpose of our evaluator is to explain how the procedures work – therefore, we ought to express compound procedure application without resorting to primitive procedure application (unless we apply primitive procedures)

[8] Ideally, user should not be able to input the tag in one's own code, as it could disrupt the operation of the evaluator, but in practice that doesn't matter much, because the semantics of Scheme does not allow to apply lists, so we can choose an arbitrary symbol as the tag, and the only undesired consequence is that the evaluator will try to evaluate expressions of the form `(('procedure-tag other-data ...) arguments)`

```
    (('quote literal)
     literal)

    (('if condition consequent alternative)
     (if (value condition environment)
         (value consequent environment)
     ;else
         (value alternative environment)))

    ((operator . operands)
     (let ((procedure (value operator environment))
           (arguments (map (lambda (operand)
                             (value operand environment))
                           operands)))
         (application procedure arguments)))

    (_
     (cond ((symbol? expression)
            (lookup expression environment))
           ((number? expression)
            expression)
           (else
            (error 'unrecognized-expression expression))))
    ))
```

The most important thing that is left is to explain how the procedure application occurs: first, we should extend the original environment of the procedure with the values of arguments, and then execute the program defined in the body of the `lambda` form. (The only exception concerns the primitive functions, which are going to be handled using the primitive `apply`)

```
(define (application procedure arguments)
  (match procedure
    (('procedure-tag parameters body closure)
     (let ((environment `(,(tie parameters arguments) . ,closure)))
       (value body environment)))
    (_ ; a primitive procedure
     (apply procedure arguments))))
```

2.4.2 Representing environments

As it was suggested in the `application` procedure, we represent the environment as a list of *frames*, where each frame is a list of two-element `(key value)` lists[9].

[9] Traditionally, frames would rather be represented as lists of dotted pairs of the form `(key . value)` (so-called *assoc lists*), because it would allow to represent a single entry us-

```
(define (lookup-frame name frame)
  (match frame
    (((key value) . rest)
     (if (equal? key name)
       `(,key ,value)
     ;else
       (lookup-frame name rest)))
    (()
     #false)))

(define (lookup name environment)
  (let (((frame . frames) environment))
    (match (lookup-frame name frame)
      ((key value)
       value)
      (_
       (lookup name frames)))))
```

The `tie` function ties the parameter names with their values, extending a given environment. Although it is not required by the Scheme standard, the `tie` function can be defined to support destructured bindings:

```
(define (tie names values)
  (match names
    (()
     '())
    ((name . other-names)
     (let (((value . other-values) values))
       `(,@(tie name value) . ,(tie other-names other-values))))
    (rest
     `((,rest ,values)))))
```

The initial environment should contain all the primitive functions available in the language. In the simplest case, it should be sufficient to define it as

---

ing a single cons cell, whereas the `(key value)` list normally uses two cons cells. However, since data structure optimization is the core theme of this work, we don't value such arguments too much. We believe that improper lists should only be used for constructing and destructuring lists, and that actual data should only be stored using proper lists (unless the data stored represents the Scheme source. However, even on such occasions we could treat the dot as a special symbol, rather than a part of syntax). Note also, that another popular representation of frames consists of two lists, where the first contains symbols' names, and the second consists of the corresponding values, for example `((a b c) (1 2 3))`. This representation is often called *a rib cage*[18].

```
(define initial-environment
  `(((cons ,cons)
     (car ,car)
     (cdr ,cdr)
     (eq? ,eq?)
     (pair? ,pair?)
     (number? ,number?)
    ;; ...
    )))
```

to use the underlying representation of primitive data structures. In the following chapter we will try to explain how the primitive data structures can be implemented.

3

The computation model and the target language

In the previous chapter, we've expressed the subset of Scheme of our interest in this very subset. We used the representations of symbols, numbers and lists provided by the host Scheme implementation.

However, this level of detail is insufficient for our purpose, because it does not reflect the capabilities and limitations of real machines.

In particular, the typical computer architectures consist of registers and an array of memory cells that can hold integer numbers from some limited range.

3.1 Memory model and primitive data

3.1.1 Memory allocation

This memory model can be expressed using the *byte vector* data type that is present in the standard Scheme[1]. Note that at this stage our evaluator is no longer "meta-circular", as the subject language becomes different from the object language. The significance of this distinction is twofold. The strict order of evaluation and state mutations are the traits of the Scheme programming language that it shares with machine languages of popular architectures. However, we employ it here only to explain certain classes of

[1] Admittedly, the choice of byte vectors as a base of our memory does impose a restriction on the memory model, namely – that the size of a machine word must be a multiple of eight bits. However, the reality shows that this the sort of restriction that one can live with. The readers who are worried with this attachment to the number eight should be calmed that it has nothing to do with the merits of this work, and the main reason for this choice is to make the considerations contained here more realistic. The key point is that the size of a single memory cell is limited (unlike, for instance, the size of a number representable in Scheme, which in particular cases could even span over thousands of clusters, should that ever be needed).

optimizations that we wish to rely on while programming in the previously described subset of Scheme, as both strictness and mutability are hostile to comprehension and software compositionality.

We can implement the allocator using a byte vector that will represent the random access memory of our machine, and a variable that would point to the free area of the memory. Since memory is just a large array of bytes, we would like to be able to impose a certain structure on it (pretty much like it is done using `struct`s in the C programming language or `record`s in Pascal). We can represent a structure using a list of `(<name> <type>)` pairs, for example, to say that a `cons` cell consists of two fields, `left` and `right`, and the size of each of those fields is the size of a machine word, we could write:

```
(define pair
  '((left word)
    (right word)))
```

Knowing a `MACHINE-WORD-SIZE`[2], we could check the size of a structure using the `size` function:

```
(define (size struct)
  (match struct
    (()
     0)
    ('word
     MACHINE-WORD-SIZE)
    (((name type) . rest)
     (+ (size type) (size rest)))))
```

This would be very uninteresting, though, if we were unable to refer to a specific field in a structure, i.e. to actually set or get a value at a certain address. To do so, we need to be able to retrieve an offset of a given field:

```
(define (offset field struct)
  (let ((prefix (take-while (lambda ((name type))
                              (not (eq? name field)))
                            struct)))
    (size prefix)))
```

Of course, the procedures for getting and setting values of memory cells would need to operate in the context of some memory object:

```
(define memory (make-bytevector MEMORY-SIZE))
```

[2]We assume here that the size of a machine word is expressed in the number of bytes.

```
(define (set-struct-field-at! location struct field value)
  (bytevector-set! (size (type field struct))
                   (+ location (offset field struct))
                   value))

(define (struct-field-at location struct field)
  (bytevector-ref (size (type field struct))
                  (+ location (offset field struct))))
```

where `bytevector-ref` and `bytevector-set!` are defined in the core Scheme[3].

Note that although neither of the above procedures supports nested structures, it would be straightforward to add such support.

Given these auxiliary procedures, we can now take a look at how the memory is allocated. We need to have a pointer that would indicate the area of memory that hasn't been allocated, and that would increase with each allocation, until it would reach the end of memory.

```
(define unallocated-memory 0)
```

The definition of the allocation procedure – i.e. the classical interpretation of the `cons` – is rather straightforward – it just allocates memory and initializes fields with given values:

```
(define (allocate-pair! left right)
  (define (not-enough-memory?)
    (is (+ unallocated-memory (size pair)) > MEMORY-SIZE))

  (if (not-enough-memory?)
    (collect-garbage! memory))

  (if #;still (not-enough-memory?)
    (error 'not-enough-memory))

  (let ((pair-location unallocated-memory))
    (set-struct-field-at! pair-location pair 'left left)
    (set-struct-field-at! pair-location pair 'right right)
    (set! unallocated-memory (+ unallocated-memory (size pair)))
    pair-location))
```

[3]This isn't quite true. In fact, the R[6]RS standard defines a family of functions that operate on typical sizes of machine words (where a machine word is a multiple of 8 bits, and "multiple" means some low power of two)[70]. These considerations are not essential for the ideas presented here, so they were simplified a bit.

It is apparent that the procedure checks whether the amount of available memory is sufficient, and it attempts to collect garbage otherwise. In the sequel, we shall ignore this issue and assume that the amount of memory is sufficient to perform any desired computation. After all, our goal is to constrain the memory usage at the earliest possible stage of program execution.

3.1.2 Primitive data types

As it was presented earlier, there's quite a few primitive data types available in Scheme, like cons-cells, numbers, symbols, booleans and procedures. On the other hand, a registry machine only allows us to hold a finite range of integer numbers in a single registry or a memory cell.

A traditional approach taken by Lisp systems was to sacrifice a few bits of a registry or a memory cell to store the information regarding the type of the object held in that cell[4].

And while this "context-free" approach is inevitable in the most general case (that is, if we want our programs to process the full spectrum of s-expressions), it is often very wasteful, as there is a whole family of functions that operate on a narrower range of objects.

Another approach, typically taken by languages such as C, is to leave the interpretation of the registry contents entirely up to the context. In such circumstances, it is the responsibility of the caller to know and satisfy the expectations and limitations of a called function, and there are specialized tools called "type checkers" that ensure this.

Eventually, everything boils down to manipulating the contents of the memory through registers. The contents of the registry can be interpreted either as the immediate data (in the case of objects such as booleans or small integer numbers or floating point numbers) or as addresses (pointers) of some objects that are too large to fit in a single memory cell.

3.2 Machine language and model

Since our goal is to construct a program that takes a functional description of a transformation and produces a sequence of machine instructions that implement this functional description, it is essential to specify the set of available machine instructions.

Of course, the instruction set does not need to be as detailed as it is for the real machines, because that would only obscure the bigger picture. Yet it has to be complex enough to be representative for actually existing machines.

[4]For more details on this approach, see for example [36] or [51].

We are going to assume that, apart from an array of heap `memory` described in the previous section, the machine also has a separate storage area for the control `stack`, and that it also has an area of (read-only) `program` memory, with a special register called `next-instruction`, which points to the instruction to be executed in the next step of the computation.

Furthermore, it has a set of general-purpose registers.

3.2.1 The instruction set

The following instructions are legal:

- `[target-register] <- source-register` — copy the contents of the `source-register` to the memory cell whose address is contained in the `target-register`[5];
- `target-register <- [source-register]` — copy the contents of the memory cell pointed to by the `source-register` to the `target-register`;
- `target-register <- register/value` — copy the value (either immediate or contained in the register on the right hand side) to the `target-register`;
- `goto register/value` — same as `next-instruction <- register/value`;
- `if left` *`cmp`* `right goto address` — compare the contents/value of `left` with `right` using *`cmp`* (where *`cmp`* can either be `<`, `<=`, `=`, `>=`, `>` or `<>`, meaning that the value of `left` is *smaller than*, *smaller than or equal to*, *equal to*, *greater than or equal to*, *greater than* or *different from* the value of `right`, respectively), and if the condition is satisfied, jump to the instruction at `address` in the program memory. Both `left`, `right` and `address` can either be a register or an immediate value (although there is little sense in left and right being both immediate values at the same time);
- `push register/value` — increases the `stack-top` by the size of a single machine word and copies the value (either immediate or contained in a register) into the memory cell pointed to by `stack-top`;
- `pop register` — moves the value from the address pointed to by `stack-top` and places it in `register`, and decreases the value of `stack-top` by the size of a single machine word;

[5] We make use of the fact that Scheme does not distinguish between round and square brackets, as long as they are paired properly.

- `register-c <- register/value-a` *`op`* `register/value-b` — assigns `register-c` to contain the result of the binary operation *`op`* on values in `register/value-a` and `register/value-b` (where *`op`* can either be `+`, `-`, `*`, `/`, `%`, `&&`, `||`[6], `^`, `<<` or `>>`, meaning the *sum*, the *difference*, the *product*, the *quotient*, the *remainder*, *bit-wise and*, *bit-wise or*, *bit-wise xor*, *arithmetic shift left* and *arithmetic shift right*, respectively);
- `halt` – causes the computation to halt.

3.2.2 A virtual machine

As before, our description can be stated more rigorously and concisely, and in a manner that is operationally valuable. Note that unlike in the previous chapter, we make an extensive use of the state mutation. Of course, rewriting the code to the functional form would be a straightforward task.

The interpretation of the instructions is performed by the `execute` function, which translates them to the actual operations to be performed by our machine. For its own purposes, the function defines four helper functions: `next`, which advances the instruction pointer and continues the computation; `fetch-instruction`, which returns the next instruction to be executed; `goto`, which modifies the instruction pointer to a given value and continues the computation from there on; and `value`, which decides whether its argument is an immediate value or a register, and in the latter case it returns the value contained in the register.

We use the Scheme symbols to represent registers

```
(define (register? register/value)
  (symbol? register/value))

(define (execute program machine)

  ;; auxiliary definitions:

  (define (next)
    (machine 'set-next-instruction!
             (+ (machine 'next-instruction) 1))
    (execute program machine))

  (define (fetch-instruction)
    (vector-ref program (machine 'next-instruction)))
```

[6] We chose a double stroke (`||`) to signify bit-wise or, because a single stroke is not a proper symbol in Scheme. The double ampersand (`&&`) was chosen for consistency of notation. Admittedly, those symbols can be confusing to people who are accustomed to the C programming language, where these "doubled" symbols are used to denote logical disjunction and conjunction rather than bit-wise operations, which are denoted using single `|` and `&`.

```
(define (goto address)
  (machine 'set-next-instruction! address)
  (execute program machine))

(define (value operand)
  (if (register? operand)
      (machine 'value-in operand)
      operand))
```

The body of the `execute` function consists of a dispatch over the current instruction, which explains what has been said earlier.

```
;; the body of 'execute' begins here:

(let ((instruction (fetch-instruction)))
  (match instruction
    (([target-register] '<- register/value)
     (machine 'set-memory-at!
              (machine 'value-in target-register)
              (value register/value))
     (next))

    ((target-register '<- [source-register])
     (machine 'set-value-in! target-register
              (machine 'memory-at (value source-register)))
     (next))

    ((target-register '<- register/value)
     (machine 'set-value-in! target-register
                             (value register/value))
     (next))

    ((target-register '<- operand-1 op operand-2)
     (let ((result (operation op (value operand-1)
                                 (value operand-2))))
       (machine 'set-value-in! target-register result)
       (next)))

    (('goto address)
     (goto address))

    (('if left cmp right 'goto address)
     (let ((result (compare cmp (value left) (value right))))
       (if result
           (goto (value address))
       ;else
           (next))))
```

```
((’push register/value)
 (machine ’push! (value register/value))
 (next))


((’pop register)
 (machine ’set-value-in! register (machine ’pop!))
 (next))


((’halt)
 machine)
))) ;; the definition of ‘execute’ ends here
```

The `operation` forms a subsystem that is traditionally called “the Arithmetic-Logic Unit” (ALU) of our machine. One can observe that we use the normal Scheme arithmetic operations to deal with numbers (however, unlike in Scheme, the range of numbers that can be processed by our machine is limited by the machine word size, and if the results of the arithmetic operations exceed that range, they will be silently truncated by our machine). As to logical operations on bits, since the standard Scheme does not provide them[7], we resort to the *Scheme Request For Implementation 60* extension[47].

```
(define (operation op operand-1 operand-2)
  (match op
    (’+ (+ operand-1 operand-2))
    (’* (* operand-1 operand-2))
    (’- (- operand-1 operand-2))
    (’/ (quotient operand-1 operand-2))
    (’% (modulo operand-1 operand-2))
    (’&& (bitwise-and operand-1 operand-2))
    (’|| (bitwise-ior operand-1 operand-2))
    (’^ (bitwise-xor operand-1 operand-2))
    (’<< (arithmetic-shift operand-1 operand-2))
    (’>> (arithmetic-shift operand-1 (- operand-2)))))
```

The `compare` function also resorts to the built-in numerical comparison predicates of Scheme:

[7] Actually, the SRFI-60 functions were eventually included in the `(rnrs arithmetic bit-wise (6))` library specified in the R^{6}RS document[70].

```
(define (compare comparison a b)
  (match comparison
    ('< (< a b))
    ('<= (<= a b))
    ('= (= a b))
    ('>= (>= a b))
    ('> (> a b))
    ('<> (not (= a b)))))
```

What is left is to explain how a machine is constructed. In our case, it is apparent that the machine is a stateful procedure (a closure) that accepts the commands `next-instruction`, `set-next-instruction!`, `value-in`, `set--value-in!`, `memory-at`, `set-memory-at!`, `push!` and `pop!`. It should be clear that all these procedures are trivial in that all that they do is retrieve or modify the contents of machine memory.

The procedure `make-machine` creates a new machine with specified parameters:

```
(define (make-machine registers memory)
  (let ((registers (map (lambda (register)
                          `(,register . 0))
                        `(next-instruction stack-pointer
                                           . ,registers)))
        (stack '()))

    (define (this-machine . command)
      (match command
        (('set-memory-at! address value)
         (let ((bytes (machine-word-bytes value)))
           (for offset in 0 .. (- MACHINE-WORD-SIZE 1)
                (bytevector-u8-set! memory (+ address offset)
                                    (list-ref bytes offset)))))

        (('memory-at address)
         (let ((bytes (map (lambda (i)
                             (bytevector-u8-ref memory i))
                           (range 0  (- MACHINE-WORD-SIZE 1)))))
           ((number/base 255) bytes)))

        (('set-value-in! register value)
         (let ((memory-cell (assoc register registers)))
           (set-cdr! memory-cell value)))

        (('value-in register)
         (let (((register . value) (assoc register registers)))
           value))
```

```
        (('next-instruction)
         (this-machine 'value-in 'next-instruction))

        (('set-next-instruction! value)
         (this-machine 'set-value-in! 'next-instruction value))

        (('push! value)
         (set! stack (cons value stack)))

        (('pop!)
         (let (((top . below) stack))
           (set! stack below)
           top))
        ))

    this-machine))
```

Note that we used the `for` control structure and the `range` function. Although their meanings should be intuitive to the reader, they are not a part of the standard Scheme, so they are defined in the appendix A, just like the functions `number/base` and `machine-word-bytes`.

3.2.3 A sample program

In order to execute a program on a machine, we need to have a program and a suitable machine.

A simple program that computes a factorial function could look like this:

```
(define factorial
  '#((n <- 5)            ;0
     (acc <- 1)          ;1
     (if n = 0 goto 6)   ;2
     (acc <- acc * n)    ;3
     (n <- n - 1)        ;4
     (goto 2)            ;5
     (halt)              ;6
     ))
```

It uses exactly two registers, called `n` and `acc`. It does not use any stack nor memory cells, so it can run on a machine with no memory (other than the registers):

```
(define tiny-machine (make-machine '(n acc) (make-bytevector 0)))
```

In order to run the program on the machine, one simply has to type in

```
(execute factorial tiny-machine)
```

After the computation terminates, the `acc` register contains the result (which should be retrievable using `(tiny-machine 'value-in 'acc)` command).

3.2.4 Assembler

The machine code for computing the factorial function from the previous section was written in a highly non-composable style, because it contained instructions such as `(if n = 0 goto 6)` or `(goto 2)` – adding a single instruction at the beginning of the program would ruin the logic of the program.

For this reason, it is convenient to introduce labels to mark certain entry points to the program. To represent the labels, we are going to use the extension to Scheme known as *keywords*, as defined, in the *Scheme Request For Implementation 88*[26] document[8], because they do not interfere with our decision to use symbols to denote registers.

The program for computing the factorial expressed in this position-independent way could look as follows:

```
'((n <- 5)
  (acc <- 1)
factorial:
  (if n = 0 goto end:)
  (acc <- acc * n)
  (n <- n - 1)
  (goto factorial:)
end:
  (halt))
```

We need a processor that would transform this form of programs into the actual machine instructions:

```
(define (assemble position-independent-code)

  (define (positions+labels (line labels) instruction)
    (if (keyword? instruction)
      `(,line ((,instruction . ,line) . ,labels))
    ;else
      `(,(+ line 1) ,labels)))
```

[8] In short, keywords are similar to symbols, but they always evaluate to themselves (like numbers) and hence cannot be bound to any value. Their distinctive characteristic is that they end with a colon.

```
(let* (((_ labels) (fold-left positions+labels
                              '(0 ())
                              position-independent-code))
       (instructions (filter (lambda (line)
                               (not (keyword? line)))
                             position-independent-code))
       (assembled (map (lambda (instruction)
                         (tree-map (lambda (item)
                                     (if (keyword? item)
                                       (assoc-ref labels item)
                                     ;else
                                       item))
                                   instruction))
                       instructions)))
  (list->vector assembled)))
```

where

```
(define (tree-map proc tree)
  (map (lambda (item)
         (if (pair? item)
             (tree-map proc item)
         ;else
             (proc item)))
       tree))
```

4

Compilation

In the previous two chapters, we have shown the source language that we wish to express our programs in, and the target language that models the machine code that is actually used by the real computers to perform computations.

In a way, those two languages are the complete opposites of each other: the first one is about composing functions, allows no side effects such as assignment and provides an implicit memory model. The other makes memory operations explicit, allows to exchange information solely with the use of assignment, and the only thinkable way of performing composition is by sequencing operations and subprograms.

The transformation from the first sort of languages to the second has traditionally been called *compilation*, and some of its popular techniques will be presented in this chapter.

We shall begin with transforming Scheme programs into a special form that does not contain any nested function calls, and hence should be easier to transform to the program on our machine.

4.1 Continuation-Passing Style

Consider the following procedure for computing the coefficient $\Delta = b^2 - 4ac$ that is helpful in finding the roots of a quadratic equation $ax^2 + bx + c = 0$:

```
(define (delta a b c)
  (- (* b b) (* 4 (* a c))))
```

Prior to computing the value of the whole expression, we need to have our machine compute the values of sub-expression and store them somewhere.

Provided that our machine has a sufficient number of registers, we could expect it to compile to the following sequence of machine instructions[1]:

[1]Note that we use a new register to hold the result of each intermediate computation. Of

```
(bb <- b * b)
(ac <- a * c)
(4ac <- 4 * x2)
(bb-4ac <- x1 - x3)
```

Let us now ask the opposite question: given a sequence of machine instructions, how can we express them in Scheme (or λ calculus)?

We typically imagine that a von Neumann machine operates by altering its current state (and indeed, this is how we implemented our virtual machine in Scheme).

However, we could imagine that there is something quite different going on: each assignment to a register can be perceived not as actually altering some value, but as creating a new scope where the original variable has been shadowed with a new one (bound with the altered value), and where *the rest of the program* is evaluated.

In order to clarify things a bit, we can define an auxiliary function `pass` that takes a value and a procedure and simply passes the value to the procedure:

```
(define (pass value procedure)
  (procedure value))
```

This way, we could rewrite programs like

```
(bb <- b * b)
... the rest of the program ...
```

as

```
(pass (* b b)
  (lambda (bb)
    ... the rest of the computation ...))
```

The procedure that represents the rest of the computation has traditionally been called a *continuation*, and the form of a program where control is passed explicitly to continuations is called *continuation-passing style*[73].

If we were to define our `delta` procedure using the continuation-passing style, we would need to extend its argument list with a continuation, i.e. a parameter that would explain what to do next with the value that our function has computed. Moreover, we could demand that all the functions that we use behave in the same way, i.e. that instead of the function `pass`,

course, the real machines usually have a limited number of registers, but the assumption that each register is assigned exactly once leads to the form of programs called *Static Single-Assignment form* (or SSA for short), which is used, for example, in the machine language of the LLVM virtual machine.

we would have functions `pass*` and `pass-` that would compute the values of operations `*` and `-` and pass them to their continuations. This way we make sure that there are no nested expressions in our program.

```
(define (pass-delta a b c continuation)
  (pass* b b
    (lambda (bb)
      (pass* a c
        (lambda (ac)
          (pass* 4 ac
            (lambda (4ac)
              (pass- bb 4ac
                (lambda (bb-4ac)
                  (continuation bb-4ac)))))))))))
```

It should be rather clear that this form of representing programs isn't particularly handy, and should only be used as an intermediate or transient representation of a program.

Of course, the above program is rather straightforward, as it consists only of a sequence of applications of primitive functions. To make the correspondence between continuation-passing style and normal Scheme programs more complete, we need to consider general function applications and conditional expressions.

Let us consider the program that computes an *absolute value* of a number. In Scheme, we would define it as

```
(define (abs n)
  (if (is n >= 0)
    n
    (- n)))
```

The corresponding machine code would look more or less like this:

```
(if n >= 0 goto 2)
(n <- 0 - n)
;; the rest of the program
```

which in turn loosely corresponds to the following continuation-passing function

```
(define (pass-abs n continuation)
  (if (>= n 0)
    (continuation n)
    (pass- 0 n continuation)))
```

The above code was trivial in that it used the condition that is directly representable in our machine code. However in general we can place arbitrarily nested Scheme expressions as the `if`'s `<condition>` clauses.

Continuations can also be used for returning multiple values. For example, the code finds the roots of a quadratic equations would need to check whether the discriminant Δ is non-negative in order to proceed with the computation of the roots:

```
(define (quadratic-roots a b c)
  (cond ((is (delta a b c) > 0)
         (values (/ (- (- b) (sqrt (delta a b c))) (* 2 a))
                 (/ (+ (- b) (sqrt (delta a b c))) (* 2 a))))
        ((is (delta a b c) = 0)
         (/ (- b) (* 2 a)))))
```

Note that we have used the `values` form that is used in Scheme for returning multiple values. Although we didn't introduce it to be the part of our host language, its meaning in the context of the discussion regarding continuation-passing style is obvious (we simply pass more than one value to the continuation). Of course, we could have instead returned a list of values, which in turn would force us to fix on some particular representation of lists, which we want to avoid at this moment.

Note also, that we defined `quadratic-roots` to return meaningful values only if the `delta` is either `zero?` or `positive?` – that is, if it is negative, then the expression `quadratic-roots` has no values (or in other words, its value is *unspecified*).

Lastly, some readers may find it displeasing, that we didn't capture the value of `(delta a b c)` using the `let` form (which is something that we would normally do to avoid some redundant computations, and more specifically, not to repeat ourselves). We ask those readers to be forgiving, as our goal at this point is to demonstrate the correspondence between various Scheme programs and their CPS counterparts, rather than promote good programming practices. Or in other words, we are in the position of a surgeon performing an operation on a patient. Of course, it is in general good for health to jog, but it would be insane to recommend jogging to someone who is lying on an operation table with open veins.

The computation of `quadratic-roots` is a bit tricky

```
(define (pass-quadratic-roots a b c continuation)
  (pass-delta a b c
    (lambda (delta#1)
      (if (> delta#1 0)
        (pass-delta a b c
          (lambda (delta#2)
            (pass-sqrt delta#2
              (lambda (sqrt@delta#3)
                (... (continuation -b-sqrt@delta/2a#6
                                   -b+sqrt@delta/2a#11)
                     ...)))))
      ;else
        (pass-delta a b c
          (lambda (delta#12)
            (if (= delta#12 0)
              (pass- 0 b
                (lambda (-b#13)
                  (pass* 2 a
                    (lambda (2a#14)
                      (pass/ -b#16 2a#14
                        (lambda (-b/2a#15)
                          (continuation -b/2a#15)))))))
            ;else
              (continuation)))))))
```

Some (rather trivial) parts of the code for computing roots were omitted for clarity. It should be clear now that for complex conditions we simply compute the value of a condition, and then pass it to a continuation that takes the result and, depending on its value, either executes the CPS version of its `<then>` branch or the CPS version of its `<else>` branch.

Note also, that – in order to avoid accidental name clashes – we generated a new name for the result of each evaluated (or executed) expression.

Let's now consider the following definition of the `factorial` function:

```
(define (factorial n)
  (if (= n 0)
      1
  ;else
      (* n (factorial (- n 1)))))
```

We can imagine that its continuation-passing version could look like this:

```
(define (pass-factorial n continuation)
  (if (= n 0)
    (continuation 1)
  ;else
    (pass- n 1
      (lambda (n-1)
        (pass-factorial n-1
          (lambda (n-1!)
            (pass* n n-1!
              (lambda (n*n-1!)
                (continuation n*n-1!)))))))))
```

The questions that we need to ask are: (1) how do we transform arbitrary functional Scheme code to continuation passing style and (2) how do we transform continuation passing style program to machine code.

4.2 Conversion to Continuation-Passing Style

The meta-circular evaluator presented in chapter 2 performed case analysis on the shape of expression to be evaluated, and had to consider six cases: `lambda` form, `quote` form, `if` form, function application, symbols and numbers. It did not deal with the `define` form, as it could be expressed using `lambda` and fixed point combinators.

For the purpose of conversion to continuation-passing style, we shall consider the `define` forms as well, because although they shouldn't be strictly necessary, the implementation of recursion in machine code is rather straightforward (certainly more so than of fixed point combinators).

The core of the transformation is the `passing` function, which takes an expression and produces its continuation-passing counterpart.

It takes an additional argument, a continuation expression, to which it shall pass the value of the transformed expression.

In case of quoted values, it only invokes the continuation:

```
(define (passing expression continuation)
  (match expression
    (('quote _)
     `(,continuation ,expression))
```

The case of conditional expression is a bit trickier: we need to pass the value of a *condition* to a new continuation, which – depending on that value – passes either the value of the *then* branch, or the value of the *else* branch to the original continuation. Note that we need to provide an original name for the result of the condition, to avoid accidental name clashes:

```
((’if <condition> <then> <else>)
 (let ((result (original-name <condition>)))
   (passing <condition>
     ‘(lambda (,result)
        (if ,result
            ,(passing <then> continuation)
            ,(passing <else> continuation))))))
```

The continuation-passing version of the `lambda` form should receive additional argument – a continuation – and the body should be converted to the continuation-passing style[2]. Since procedures are first-class values, they shall be passed to the original continuation just like quoted values.

```
((’lambda <args> <body>)
 ‘(,continuation (lambda (,@<args> return)
                   ,(passing <body> ’return))))
```

Function application is the trickiest bit: we need to compute the values of all the compound arguments, passing them to the subsequent continuations, which finally invoke the continuation-passing version of the called function (obtained using the `passing-function` function), passing its result to the original continuation[3]

```
((function . arguments)
 (let ((simple-arguments (map (lambda (argument)
                                (if (compound? argument)
                                    (original-name argument)
                                ;else
                                    argument))
                              arguments)))
   (passing-arguments arguments simple-arguments
                      ‘(,(passing-function function)
                        ,@simple-arguments
                        ,continuation))))
```

Otherwise, the expression is just a value to be passed to the continuation:

```
 (_
  ‘(,continuation ,expression))))
;; the definition of ‘‘passing’’ ends here
```

[2]Although in this work we have been consequently passing the continuation as the last argument for the purpose of clarity, in practice it might be a better idea to make it the first argument, because that would allow to handle variadic functions properly.

[3]Note that, for clarity of presentation, we depart from the definition of Scheme in that we do not allow complex expression in the head (function) position.

The `passing-arguments` helper function (used for dealing with applications) takes three arguments: a list of arguments to the called function, a list of new names for the compound arguments, and a final call to be made from the nested chain of continuations:

```
(define (passing-arguments arguments names final)
  (match arguments
    (()
     final)

    ((argument . next)
     (let (((name . names) names))
       (if (compound? argument)
           (passing-arguments next names
                              (passing argument
                                `(lambda (,name) ,final)))
       ;else
           (passing-arguments next names final))))))
```

The `passing-program` function takes a program, that is, a sequence of definitions and an expression, and converts each of the definitions to the continuation-passing style. For the sake of simplicity, we shall assume here, that all the definitions are function definitions. Note that we need to pass the additional `return` argument that is stripped away after the conversion.

We assume, that the program passes its result to the `exit` continuation.

```
(define (passing-program program)
  (let ((((’define names functions) ... expression) program))
    `(,@(map (lambda (name function)
               (let ((’return pass-function) (passing function
                                                      ’return)))
                 `(define ,(passing-function name)
                          ,pass-function)))
             names functions)
      ,(passing expression ’exit))))
```

The complete code, with the implementations of `passing-function` and `original-name` can be found in the appendix E.

We can check, that the value of

```
(passing-program ’((define !
                     (lambda (n)
                       (if (= n 0)
                           1
                           (* n (! (- n 1))))))
                   (! 5)))
```

is the form

```
((define pass-!
   (lambda (n return)
     (pass= n 0
       (lambda (n=0/1)
         (if n=0/1
            (return 1)
         ;else
            (pass- n 1
              (lambda (n-1/3)
                (pass-! n-1/3
                  (lambda (!/n-1/2)
                    (pass* n !/n-1/2 return)))))))))
 (pass-! 5 exit))
```

4.3 Generating machine code

Usually, function serves as an *abstraction barrier* in complex systems: we sometimes imagine it as a black box that for some given input it produces some output. As such, functions are often perceived as *compilation units*: a single function corresponds to a distinguished block of compiled code, along with its entry point.

When designing an interface for the abstraction barrier on an actual system, we need to answer the following questions:

- How are the parameters passed on to a function?
- How are the values returned from a function?

On our machine, the possible answers are that arguments and values can either be passed through registers, through stack or through the memory heap.

Typically, passing values through registers is most efficient and therefore most desirable. However, since the number of registers in a CPU is usually small, some other conventions often need to be established (for example, the first few arguments can be passed through registers, and another ones through the stack or heap).

Another question is, how can a function know where the control should be transferred after it finishes its execution. Typically, this information is stored on the *call stack*, which stores the appropriate address in the caller code.

However, while the use of stack is in general inevitable, sometimes it may be more desirable to store the return address in a register, and only save it

when invoking another function (because this can decrease the number of memory accesses, which are typically more expensive than register access).

The latter option, although may seem less obvious, allows to perceive function calls as gotos that pass arguments[73], where the return address is just another argument to be passed.

The advantage of this approach is that if a call to another function is the last thing that a calling function does, then the called function can simply inherit the return address from the caller. This trick is called *Tail Call Optimization* and allows, among other things, to express loops using recursion.

Let's recall our continuation-passing style version of the factorial function:

```
(define pass-!
  (lambda (n return)
    (pass= n 0
      (lambda (n=0/1)
        (if n=0/1
           (return 1)
        ;else
           (pass- n 1
             (lambda (n-1/3)
               (pass-! n-1/3
                 (lambda (!/n-1/2)
                   (pass* n !/n-1/2 return/1)))))))))
```

The function is recursive, but it is not tail recursive, because it calls another continuation from within the recursive call.

We expect it to be transformed to something similar to the following assembly code:

```
factorial:
  (if n <> 0 goto else:)
  (result <- 1)
  (goto return)
else:
  (n-1 <- n - 1)
  (push n)
  (push return)
  (n <- n-1)
  (return <- proceed:)
  (goto factorial:)
proceed:
  (pop return)
  (pop n)
  (n-1! <- result)
  (n*n-1! <- n * n-1!)
  (result <- n*n-1!)
  (goto return)
```

In addition to registers corresponding to the continuation arguments (i.e. `n`, `n-1`, `n-1!` and `n*n-1!`), there are two additional registers – `return`, which stores the return address of current procedure, and `result`, which is used for passing the function's result to the caller.

One can see that, prior to the recursive call, we had to store the return address on the stack and then restore it after the return from the call. We also had to save and restore the value of the `n` register, because it was used as an argument to the `factorial:` procedure, but its original value was used in the sequel of the procedure.

Let's now consider the tail-recursive version of the procedure, which takes an additional argument – the accumulator – to store the result:

```
(define !+
  (lambda (n a)
    (if (= n 0)
        a
    ;else
        (!+ (- n 1) (* n a)))))
```

Its continuation-passing version looks like this

```
(define pass-!+
   (lambda (n a return)
     (pass= n 0
       (lambda (n=0/1)
         (if n=0/1
           (return a)
         ;else
           (pass* n a
             (lambda (n*a/3)
               (pass- n 1
                 (lambda (n-1/2)
                   (pass-!+ n-1/2 n*a/3 return)))))))))
```

which in turn roughly corresponds to the following assembly code:

```
  factorial+:
    (if n <> 0 goto else:)
    (result <- a)
    (goto return)
  else:
    (n*a <- n * a)
    (n-1 <- n - 1)
    (n <- n-1)
    (a <- n*a)
    (goto factorial+:)
```

The code does not perform any stack operations, and it is clear that the function call is performed just as a simple goto with register assignment.

Note that the calling function has to know the names of the registers that are used to pass arguments to the called function. It may also have to know what registers are used by the called function internally (including the registers used by all functions that are called by the called function, as well as registers used by the functions called by these functions, and so on) in order to know whether it should save their values on the stack before the call, and restore them afterwards.

Also, the code generated by our procedure is wasteful with regard to the number of used registers. Normally, computers have a limited number of registers, and compilers try to reuse them as much as possible in order to minimize the number of accesses to RAM (which is typically much slower than manipulating register values).

It would therefore be more realistic to rename the arguments to functions in a systematic way, and also minimize the number of registers that are used within a procedure (this process is called *register allocation* in the literature[84] [51]).

However, since these issues have very little to do with the merit of this work, we will proceed with our assumption, that the number of registers of our machine is sufficient to perform any computation we desire (which, at this very moment, is either computing a factorial or – ultimately – sorting an array).

We therefore assume that each calling function knows at least the names of registers for each defined procedure, that will be available via `argument--names` helper function.

```
(define (passing-program->assembly program/CPS)
  (let ((((’define names passing-functions)
          ...
          expression) program/CPS))

    (define (argument-names function-name)
      (any (lambda (name (’lambda (arguments ... return) . _))
             (and (equal? name function-name)
                  arguments))
           names passing-functions))
```

The main procedure, invoked recursively for each defined function, as well as for the main expression of the program, should be able to transform an expression in CPS form to a piece of assembly.

There are actually only three cases that we need to consider: branching, invocation of a function, and returning a value. Since – as we mentioned earlier – upon invocation, the program may need to save the information contained in the registers whose content might be overwritten by a called function – we need to track the registers used up to a given point.

The code for returning a value to the continuation is rather trivial – we assign the desired value to the `result` register and perform a jump to the address contained in the `return` register:

```
    (define (assembly expression/cps registers)
      (match expression/cps
        ((’return value)
         `((result <- ,value)
           (goto return)))
```

The code for branching is a bit tricky, as we need to undo some of the effects of our CPS transformation, to handle the conditionals properly (as we noted in chapter 2, Scheme provides Boolean values `#true` and `#false`, but here – for simplicity – we assume, that the instruction `(if a >?< b goto c)` can only be generated from the code of the form `(pass<?> a b (lambda (a<?>b) (if a<?>b ...)))`. Furthermore, to attain some readability, we

shall inverse the condition in the comparison, and perform jump to the `else` branch). We use the `sign` function, which converts names like `pass=` or `pass+` to operators like `=` or `+`.

We generate a new label for the `<else>` branch, add `a` and `b` to the set of used registers, generate a branching instruction followed by assembly for the `<then>` expression, followed by the label for the `<else>` branch, followed by machine code for the `<else>` expression.

```
((pass<?> a b ('lambda (a<?>b) ('if a<?>b
                                    <then>
                                    <else>)))
  (let ((else (new-label 'else))
        (registers (union registers
                          (maybe-register a)
                          (maybe-register b))))
    `((if ,a ,(inversion (sign pass<?>)) ,b goto ,else)
      ,@(assembly <then> registers)
      ,else
      ,@(assembly <else> registers))))
```

For clarity, invocation should be handled by a separate function:

```
    ((operator . operands)
     (call operator operands registers))))
;; the definition of ``assembly'' ends here
```

We need to take the following factors into consideration:

- whether the continuation is a `lambda` expression or a `return` expression
- whether the current operator is primitive (like `pass+`) or a defined function (like `pass-factorial` from our example), or a `lambda` expression (anonymous function)

```
(define (call operator operands registers)
  (cond ((primitive-operator? operator)
         (call-primitive operator operands registers))

        ((defined-function? operator)
         (call-defined operator operands registers))

        ((anonymous-function? operator)
         (call-anonymous operator operands registers))))
```

"Calling" a primitive operator is easy – we simply transform it to assembly instruction, followed by the code generated from the body of the continuation (or a `(goto return)` instruction in the case of a call to the `return` continuation)

```
(define (call-primitive operator operands registers)
  (let (((left right continuation) operands))
    (match continuation
      (('lambda (result) body)
       `((,result <- ,left ,(sign operator) ,right)
         ,@(assembly body (union registers
                                 (maybe-register left)
                                 (maybe-register right)
                                 `(,result)))))
      (_ ;; a ``return'' continuation
       `((result <- ,left ,(sign operator) ,right)
         (goto ,continuation))))))
```

In order to call a defined procedure, we need to save the arguments that we might be using in the future. At this stage, we could perform a fairly elaborate analysis to find out which registers that are used by our function after the call are overwritten by the called functions, and only save those.

However – again, for the sake of simplicity – we shall only check, which registers are going to be needed after we return from the call (and we only do so if there's actually anything to be done after the return – otherwise we should perform the tail call optimization).

We save registers by performing a series of `push` instruction, and restore them by executing `pop` in the reverse order.

```
(define (call-defined function arguments registers)

  (define (save registers)
    (map (lambda (register)
           `(push ,register))
         registers))

  (define (restore registers)
    (map (lambda (register)
           `(pop ,register))
         (reverse registers)))

  (define (pass values function)
    (let ((names (argument-names function)))
      (map (lambda (name value)
             `(,name <- ,value))
           names values)))
```

```
;; the body of ``call-defined'' begins here
(let (((arguments ... continuation) arguments)
      (entry (passing-function-label function)))



  (match continuation
    (('lambda (result) body)
     (let* ((proceed (new-label 'proceed))
            (sequel (assembly body registers))
            (registers (intersection registers
                                     (used-registers sequel))))
       `(,@(save registers)
         ,@(pass arguments function)
         (push return)
         (return <- ,proceed)
         (goto ,entry)
         ,proceed
         (pop return)
         ,@(restore registers)
         ,@sequel)))



    (_ ;; tail call optimization
     `(,@(pass arguments function)
       (goto ,(passing-function-label function)))))))
```

Given all these helper functions, we can now express the compilation of a whole program[4]. As noted earlier, we assume that a program is a sequence of function definitions followed by a single expression. We therefore need to compile both the definitions and the expression. Furthermore, we need to take into account what should happen after our program finishes its execution. Obviously, we want our machine to `halt`.

[4] A careful reader probably noticed that we're lacking the definitions of `anonymous-function?` and `call-anonymous`. These definitions are trivial, as the call to anonymous function boils down to register assignment followed by execution of assembly code of the body of that function. They would contribute nothing to the examples presented here, so we allowed ourselves to omit them here. They are of course available in the appendix E.

```
;; body of ‘‘passing-program->assembly’’ begins here
‘((return <- end:)
  ,@(assembly expression '())
  ,@(append-map (lambda (name ('lambda args body))
                  ‘(,(passing-function-label name)
                    ,@(assembly body '())))
                names passing-functions)
  end:
  (halt))))
;; the definition of ‘‘passing-program->assembly’’ ends here
```

We can observe that the programs for computing factorial function behave roughly as we expected them to: the tail recursive version does not perform any stack operations and only uses `goto` to transfer control. The other version saves the `return` register on the stack prior to the call, along with other registers that are needed in the sequel.

4.4 Conclusion

Although the compiler presented in this chapter successfully transforms some high level functions to efficient machine code, it is of course by no means complete. It does not handle higher order functions properly, nor does it support arbitrary precision arithmetic. Moreover, it does not perform any register allocation and uses a new register for storing each intermediate result, which makes it inapplicable to real machines. It does, however, serve its purpose, in that it gives a rough overview of the compilation process.

5

Reasoning about programs

In the previous chapter we have seen how an arbitrary Scheme program can be transformed to a particular form that has certain properties which make it suitable for execution on a sequential machine. In particular, this form specified the order of evaluation of arguments, which would otherwise be unspecified[1].

In this chapter, we will present a broader class of Scheme to Scheme transformations, called *equational reasoning*.

As the name suggests, the purpose of these transformations is *reasoning*, that is, drawing certain conclusions about programs.

Broadly speaking, we have already seen a simple example of a reasoning system, namely – the evaluator itself, which allowed us to conclude the values of expressions for given arguments.

However, this system only allowed us to conclude about some very specific properties, like, that the value of expression `(+ 2 2)` is the number `4`.

For the purpose of this work, we would like to be able to prove our claims about some more abstract properties of our program, like that the `qsort` function actually sorts a given sequence, or that at least the length of its output is the same as the length of its input, and that all the elements that were present in the input are also present in the output.

5.1 Basic terminology

Following the tradition, we shall call an expression whose logical value may be `#false` or `#true`, a *proposition* or a *sentence*.

A `lambda` expression whose application is a proposition is called a *sentential form* or a *predicate*. For example, `(lambda (x y) (= (+ x y) (+ y x)))` is a sentential form.

[1] Note that in general leaving the order of evaluation unspecified is a good thing, because it allows to conceive interesting evaluation strategies.

A sentential form that is true for all arguments is called a *theorem*. A sentential form that is suspected to be a theorem is called a *claim* or a *conjecture*.

In order to verify that a conjecture is a theorem, we need to construct a *proof*. Pragmatically, a proof is an argument whose purpose is to convince us about the validity of a certain claim. However, the advances in logic and meta-mathematics resulted in a more formal concept of a proof as a mathematical object that is constructed according to certain rules, called *rules of inference*. Typically, such proofs refer to theorems whose validity is claimed to be obvious or that have a foundational role to a theory. Such theorems are called *axioms*. For example, `(lambda (x y) (equal? (car (cons x y)) x))` may be an axiom.

Proofs may also refer to other theorems that have been proved earlier. Used in such way, those theorems are typically called *lemmas*.

Of course, the fact that we can construct proofs as mathematical objects is insufficient for us to accept their validity. First and foremost, we must accept the validity of the rules of inference proposed by a particular formal system and decide whether they conform to our intuition. Since different people may have different intuitions, we expect that there be no agreement with regard to the choice of a particular formal system, and it must inevitably be left to the reader to decide whether the "proofs" presented here are actually proofs.

There are, however, certain properties that our formal systems may or may not have, that are helpful in judging their usefulness. Logicians consider, for example, whether their formal systems are *sound* (i.e. whether their rules allow us to only prove formulas that are actually true) and *complete* (i.e. whether they allow us to prove all true formulas that can be expressed in our language). For more details, see [7].

5.2 The reasoning system

In this section, we are going to present a reasoning system based on the work of Boyer and Moore[9]. The rules and axioms are taken from [32], which is a very accessible hands-on introduction to the topic[2].

The axioms/theorems have the following form:

```
<theorem> ::= (lambda (x1 ... xn) <rule>);

<rule> ::= (equal? <expression> <expression>)
         | (if <expression> <rule> <expression>)
         | (if <expression> <expression> <rule>);
```

[2] Thank you, Daniel Friedman and Carl Eastlund.

where `x`$_1$ ... `x`$_n$ are distinct variables (symbols) and `<expression>` can be any Scheme expression (in particular, it can also be a `<rule>`).

That is, the body of a theorem contains an application of the `equal?` predicate, possibly nested in `<then>` or `<else>` branches of a series of nested `if` expressions.

The `<condition>`s of the `if` expressions of a `<rule>` are called *premises*, and the final `<rule>` consisting of the application of `equal?` predicate is called a *conclusion*.

As mentioned earlier, we differentiate between axioms and theorems in that we require no proof of the former. We shall express this difference by saying that we `assume` the axioms (or, to be exact, their validity), whereas the validity of theorems is `assured`[3].

We shall assume, that a form `(if <condition> <conclusion>)` is an abbreviation of `(if <condition> <conclusion> #true)`.

5.2.1 The core axioms

The core axioms of the system concern the `if` form and the `equal?` predicate:

```
(define (equal-same x)
  (equal? (equal? x x) #true))

(assume equal-same)

(define ((commutative? operator) x y)
  (equal? (operator x y) (operator y x)))

(assume (commutative? equal?))

(define (equal-if x y)
  (if (equal? x y) (equal? x y)))

(assume equal-if)

(define (if-true then else)
  (equal? (if #true then else) then))

(assume if-true)
```

[3] Choosing two names that only differ with a single character to denote two completely opposite notions may not seem to be a very good idea. We are drawing inspiration here from the creator of the Scala programming language, Martin Odersky, who did the same thing choosing the names `val` and `var` for declaring immutable and mutable variables, respectively. We are hoping that, since the worst ideas in Computer Science seem to also be the ones that last the longest, this work would actually turn out to be influential in some regards.

```
(define (if-false then else)
  (equal? (if #false then else) else))

(assume if-false)

(define (if-same condition then/else)
  (equal? (if condition then/else then/else)
          then/else))

(assume if-same)

(define (if-nest-then condition then else)
  (if condition
      (equal? (if condition then else) then)))

(assume if-nest-then)

(define (if-nest-else condition then else)
  (if condition
      #true
      (equal? (if condition then else) else)))

(assume if-nest-else)
```

The rule of inference is a bit complex, so it should be instructive to see how it works before specifying it formally.

Note that the `if-nest-else` rule is formulated in a slightly strange manner, because it contains `#true` in the `if`'s `<then>` position. We suspect that this is a way of writing a negation, and that the axiom could equivalently be written as

```
(define (if-nest-else* condition then else)
  (if (not condition)
    (equal? (if condition then else) else)))
```

where the `not` function is defined as

```
(define (not x)
  (if x #false #true))
```

Or, to put it more generally, we suspect that the following sentential form that we shall call `negation-inversion` is a theorem:

```
(define (negation-inversion condition result)
  (equal? (if condition #true result)
          (if (not condition) result)))

(assure negation-inversion)
```

5.2.2 Proof of `negation-inversion`

Consider the right hand side of the `equal?` predicate in the definition of `negation-inversion`:

```
(if (not condition) result)
```

By definition of `not` and our convention regarding `if`, we can rewrite it as

```
(if (if condition #false #true) result #true)
```

Now, by the `if-same` axiom, we can transform it to the following form[4]:

```
(if condition
    (if (if condition #false #true) result #true)
    (if (if condition #false #true) result #true))
```

We can now use the `if-nest-then` to rewrite the `<condition>` of the `<then>` branch of the main `if` expression:

```
(if condition
    (if #false result #true)
    (if (if condition #false #true) result #true))
```

Likewise, we can transform the `<condition>` of the `<else>` branch of the main `if` expression using `if-nest-else`:

```
(if condition
    (if #false result #true)
    (if #true result #true))
```

We can now reduce the same `<then>` branch as before using `if-false`:

```
(if condition
    #true
    (if #true result #true))
```

And similarly apply the `if-true` to the `<else>` branch

[4] This step may seem surprising at first, because the `if-same` axiom can be used both to reduce expressions like `(if condition expression expression)` to `expression`, and to extend `expression` to `(if whatever expression expression)`, where `whatever` can be any expression of our liking. The technique of rewriting `(if condition then else)` as `(if condition (if condition then else) (if condition then else))` is called *If Lifting*[32].

```
(if condition
    #true
    result)
```

We can now substitute this result to the original context, i.e. as the right hand side of the `equal?` expression from the definition of `negation-inversion`:

```
(equal? (if condition #true result)
        (if condition #true result))
```

We see that the right hand side is identical to the left hand side, which allows us to apply the `equal-same` rule, yielding

```
#true
```

which concludes our proof.

5.2.3 The rules of inference

We have just seen an example of equational reasoning in practice. It should be relatively easy to grasp way we used the axioms `if-same`, `if-true`, `if-false` and `equal-same`. What they all share in common is that they consist only of a conclusion, i.e. an application of the `equal?` predicate. It should be clear, that they allowed us to rewrite a form matching the shape of one of the arguments to `equal?` to the form matching the shape of the other argument.

The axiom `if-nest-then` and `if-nest-else` are a bit more tricky, though, because in addition to a conclusion, they contain a premise. Therefore we were allowed apply the `if-nest-then` axiom only because there was an expression whose shape matched one of the conclusion's arguments, and this expression lied on a `<then>` branch of a matching conclusion.

We will try to express this rule formally in the Scheme programming language.

In the above proof, we have only been showing a part of the expression that was actually of a concern to us, but we should remember, that in each step we were actually rewriting a whole expression, so for example the substitiution of the definition of `not` should be written as a transformation which converts the expression[5]

```
(equal? (if condition #true result)
        (if [not condition] result))
```

into

[5] We use the square brackets here to signify focused expressions. As noted in chapter 3, Scheme reader makes no distinction between round and square brackets, as long as the opening bracket matches the shape of the closing one.

```
(equal? (if condition #true result)
        (if [if condition #false #true] result))
```

We shall therefore need some means for selecting the sub-expression that is going to be subject to our rule. In the above example, the expression `(not condition)` is the first argument to the second argument of the main expression, which could be written as `(2 1)`, and read as "take the second argument, and then take the first argument" (the operator itself is the "zeroth" argument). Such sequence of indices will henceforth be called a *path* of a sub-expression.

We could define a selector function that takes an expression and a path, and returns a sub-expression pointed to by that path[6]:

```
(define (focus expression path)
  (match path
    (()
     expression)
    ((index . next)
     (focus (list-ref expression index) next))))
```

The core function for our reasoning system should take an expression, a path to its sub-expression of our interest, and an axiom with a hint specifying how it is meant to be used, and it should return an expression with the sub-expression transformed appropriately. For example,

```
(rewrite '(equal? (if condition #true result)
                  [if (if condition #false #true)
                      result
                      #true])
         '(2)
         '(if-same condition
                   (if (if condition #false #true)
                       result
                       #true)))
```

should return the expression

[6] It should be easy to see that the `focus` function could also be defined using `fold-left` over `list-ref`:

```
(define (focus expression path)
  (fold-left list-ref expression path))
```

```
(equal? (if condition #true result)
        [if condition
            (if (if condition #false #true)
                result
                #true)
            (if (if condition #false #true)
                result
                #true)])
```

and

```
(rewrite '(equal? (if condition #true result)
                  (if condition
                     (if [if condition #false #true]
                         result
                         #true)
                     (if (if condition #false #true)
                         result
                         #true)))
         '(2 2 1)
         '(if-nest-then condition #false #true))
```

should evaluate to

```
(equal? (if condition #true result)
        (if condition
            (if #false
                result
                #true)
           (if (if condition #false #true)
                result
                #true)))
```

Note also, that the rule of inference needs to be able to access the axioms, theorems and definitions that we refer to. However, since it would be inconvenient to pass them around to the `rewrite` function, we will make use of extension to Scheme known as *parameters*[25], and have the `current-book` parameter default to the core axioms and definitions.

In our rewriting rule, we need to differentiate between theorems (including axioms) and definitions, because, in the case of the definitions, we can only replace *definiendum* with the corresponding *definiens*, while in the case of theorems, we can replace one side of the *conclusion* with the other (this distinction should make it clear why we decided to mark axioms and theorems using the `assume` and `assure` keywords):

```
(define (rewrite expression path rule)
  (if (theorem? rule)
    (rewrite-theorem expression path rule)
  ;else
    (rewrite-definition expression path rule)))
```

where `theorem?` checks in the `current-book` whether a given rule has been declared as a theorem or axiom.

When rewriting a definition, we simply substitute arguments with corresponding values in the body of `lambda` expressions:

```
(define (rewrite-definition expression path application)
  (assert (equal? application (focus expression path)))
  (let* ((function (function-name application))
         (definiens (function-body function))
         (arguments (function-arguments function))
         (values (subject application))
         (substitution (substitute arguments values definiens)))
    (replace-subexpression expression path substitution)))
```

the `function-name` retrieves the name of function referred to in an application. The `function-body` and `function-arguments` return the body and arguments of a `lambda` expression with a given name. They ought to refer to the `current-book` and their exact definitions would depend on the particular representation of a book.

The `subject` of an application is a list of all values that the function is applied to.

The `substitute` procedure could be defined in the following way:

```
(define (substitute variables values expression)
  (match expression
    (('quote _)
     expression)
    ((head . tail)
     `(,(substitute variables values head)
       . ,(substitute variables values tail)))
    (_
     (or (any (lambda (variable value)
                (and (equal? variable expression)
                     value))
              variables values)
         expression))))
```

However, in order for it to be correct, the `expression` must not contain `lambda` expressions whose argument list would contain the symbol `lambda`,

the symbol `quote` or any of the symbols contained in the `variables` list (this condition can be assured by systematic α renaming of all the bound variables of a program [73]).

The `replace-subexpression` function is defined as follows:

```
(define (replace-subexpression expression path replacement)
  (match path
    ((index . subpath)
     (let ((prefix (take expression index))
           ((subexpression . suffix) (drop expression index)))
       `(,@prefix
         ,(replace-subexpression subexpression subpath
                                 replacement)
         ,@suffix)))
    (()
     replacement)))
```

Lastly, we're set to explain how to `rewrite-theorem`.

First, we need to substitute the body of the theorem with the supplied arguments, just like we did for definitions. Then we need to find the possible conclusions of the theorem and see (1) whether they are `equal?` to any of the arguments to `equal?` and (2) whether the premises required by the theorem are satisfied at the point of the occurrence of the term in focus.

The `conclusions+premises` returns a list of tuples. of the form `[conclusion required-premises discarded-premises]`.

```
(define (conclusions+premises theorem)
  (match theorem
    (('equal? _ _)
     `([,theorem () ()]))

    (('if condition consequent alternative)
     `(,@(map (lambda ([conclusion required discarded])
                `[,conclusion
                  ,(union condition required)
                  ,discarded])
              (conclusions+premises consequent))
       ,@(map (lambda ([conclusion required discarded])
                `[,conclusion
                  ,required
                  ,(union condition rejected)])
              (conclusions+premises alternative))))

    (_
     '())))
```

(Note that the code assumes that all the derived special forms such as `and`, `or` and single-armed `if` are expanded).

For example, `conclusions+premises` of the body of `if-nest-then`, i.e.

```
(conclusions+premises
 '(if condition
      (equal? (if condition then else) then)
      #true))
```

is the list containing a single tuple:

```
([(equal? (if condition then else) then) (condition) ()])
```

and the value for the body of `if-nest-else` is also a singleton list:

```
([(equal? (if condition then else) else) () (condition)])
```

We also need to know which premises are satisfied or refuted in the context of our focus:

```
(define (premises expression path)
  (match `(,expression ,path)
    ((_ ())
     '[() ()])

    ((('if condition consequent _) (2 . subpath))
     (let (([satisfied refuted] (premises consequent subpath)))
       `[,(union `(,condition) satisfied) ,refuted]))

    ((('if condition _ alternative) (3 . subpath))
     (let (([satisfied refuted] (premises alternative subpath)))
       `[,satisfied ,(union `(,condition) refuted)]))

    ((_ (index . subpath))
     (premises (list-ref expression index) subpath))))
```

The `premises` function returns a tuple `[satisfied refuted]`. For example, the `premises` of expression

```
'(if c
     (if a
         '(2 2)
         '(2 3))
     (if b
         '(3 2)
         '(3 3)))
```

at focus `(2 2)` are `[(a c) ()]`, at focus `(2 3)` – `[(c) (a)]`, at focus `(3 2)` – `[(b) (c)]` and – at focus `(3 3)` – `[() (b c)]`.

In order to `rewrite-theorem`, we require that the required premises be a subset of the satisfied premises, and that the discarded premises be a subset of refuted premises.

For pragmatic reasons, we also require that there be only a single conclusion in a theorem that can be used to perform a rewrite (i.e. whose premises and conclusion match). Otherwise we wouldn't know which conclusion should be chosen.

```
(define (rewrite-theorem expression path rule)
  (let* ((theorem (function-name rule))
         (definiens (function-body theorem))
         (arguments (function-arguments theorem))
         (values (subject rule))
         (instance (substitute arguments values definiens))
         (term (focus expression path))
         ([satisfied refuted] (premises expression path))
         (basis (filter
                 (lambda ([conclusion required discarded])
                   (let ((('equal? left right) conclusion))
                     (and (or (equal? term left)
                              (equal? term right))
                          (subset? required satisfied)
                          (subset? discarded refuted))))
                 (conclusions+premises instance)))
         ((([conclusion _ _]) basis)
         (('equal? left right) conclusion)
         (substitution (if (equal? left term) right left)))
    (replace-subexpression expression path substitution)))
```

Although the rule is rather lengthy, its form should be straightforward to analyze. The first few lines (up to the binding containing `instance`) are actually the same as in the `rewrite-definition` function (in the programmer's craft this would suggest that they should be extracted to a separate function). The rest of the code is concerned with matching the premises from theorem with those from the context of `expression`. The variable `basis` captures the list of matching conclusions. The binding `(([conclusion _ _]) basis)` assumes that this list contains exactly one element, and if this assumption was not satisfied, it would raise an error.

5.2.4 Proof checking

The `rewrite` procedure provides us with means of performing a single inference step. We could use it to build a program that performs a proof

checking:

```
(define (verify conjecture proof)
  (equal? (fold-left (lambda (term (path rule))
                       (rewrite term path rule))
                     conjecture proof)
          #true))
```

Now we can state the proof of `negation-inversion` more formally:

```
(verify '(equal? (if condition #true result)
                 (if (not condition) result #true))
  '(((2 1)   (not condition))
    ((2)     (if-same (if condition
                          (if (if condition #false #true)
                              result
                              #true)
                          (if (if condition #false #true)
                              result
                              #true))
                      (if (if condition #false #true)
                          result
                          #true)))
    ((2 2 1) (if-nest-then condition result #true))
    ((2 3 1) (if-nest-else condition result #true))
    ((2 2 1) (if-false result #true))
    ((2 3 1) (if-true result #true))
    ((0)     (equal-same (if condition #true result)))))
```

5.3 Totality

The `rewrite-definition` function assumed that there is nothing wrong with replacing an application of a function with the body of that function, where formal arguments are replaced with values being applied to – just as in most circumstances there was nothing wrong with replacing an application of a function to some arguments with the actual value of that function for those arguments.

However, it is not obvious that a function actually has a value. Consider the following definition:

```
(define (partial x)
  (not (partial x)))
```

In an attempt of computing the value of the expression, say, `(partial partial)`, the evaluator will never terminate.

The function `partial` is not a total function, because it does not have a defined value for every argument (as a matter of fact, it doesn't have a definite value for *any* argument), and consequently, a program whose value relies on the value of `partial` function may itself have no definite value.

The Boyer-Moore system doesn't allow to expand the definitions of functions that were not proven to be total, because they could be used to prove a contradiction[32], thereby depriving the deductive system of its cognitive value. Therefore, in order to be able to `rewrite-definition` in a legitimate way, it is required that a *totality claim* for that function is proven first.

While it is impossible to provide a universal function that would claim whether a given function is total, there exist certain classes of functions for which it is possible to derive such proofs by purely mechanical means.

In the case of recursive functions it is easy to see that if an argument that is used as a base case for recursion shrinks (in some general sense) by one unit towards the base case with each recursive call, then the function will eventually reach its base case and terminate[7].

This "general sense of argument shrinking" is called a *measure* of a function, which is a function that maps arguments to natural numbers. For many arithmetic functions, a common measure is just the identity function. For functions whose arguments are structures/expressions, the measure can be defined as

```
(define (size x)
  (if (pair? x)
    (+ 1 (size (car x)) (size (cdr x)))
  ;else
    0))
```

although – since it is defined recursively – we cannot resort to that definition in proving its own totality claim, and therefore we need to assume it.

Instead, we can characterize it with the following axioms:

```
(define (natural?/size x)
  (equal? (natural? (size x)) #true))

(assume natural?/size)
```

[7] Note that, especially in lazy languages, there are functions that do not satisfy this condition, but have a definite value nevertheless. Consider, for example, the definition:

```
(define (numbers-from start)
 (cons start (numbers-from (+ start 1))))
```

While this function may call itself potentially indefinitely many times, it is total (in the domain of numbers).

```
(define (size/car x)
  (if (pair? x)
    (equal? (< (size (car x)) (size x)) #true)))

(assume size/car)

(define (size/cdr x)
  (if (pair? x)
    (equal? (< (size (cdr x)) (size x)) #true)))

(assume size/cdr)
```

The following function can be used to obtain the totality claim for any recursive function[8]:

```
(define (totality-claim name args body measure)

  (define (claim expression)
    (match expression
      ((’quote _)
       #true)

      ((’if condition consequent alternative)
       ‘(and ,(claim condition)
             (if ,condition
                 ,(claim consequent)
                 ,(claim alternative))))

      ((function . arguments)
       (if (equal? function name)
           ‘(and (< ,(substitute args arguments measure) ,measure)
                 . ,(map claim arguments))
       ;else
           ‘(and . ,(map claim arguments))))
      (_
       #true)))

  ‘(and (natural? ,measure)
        ,(claim body)))
```

For example, consider the following definition:

[8]Note however, that it does not support mutual/nested recursion, nor recursion obtained from a fixed point combinator (which itself isn’t a total function).

```
(define (append a b)
  (if (pair? a)
    (cons (car a) (append (cdr a) b))
  ;else
    b))
```

We can obtain its totality claim by evaluating

```
(totality-claim 'append '(a b)
  '(if (pair? a)
       (cons (car a) (append (cdr a) b))
       b)
  '(size a))
```

which produces

```
(and (natural? (size a))
     (and (and #true)
          (if (pair? a)
              (and (and #true)
                   (and (< (size (cdr a)) (size a))
                        (and #true)
                        #true))
              #true)))
```

Apparently, our `totality-claim` contains many redundant `(and #true)` and `#true` conditions. They can be easily removed by expanding `and` to `if` and applying the `if-true` axiom:

```
(if (natural? (size a))
    (if (pair? a)
        (< (size (cdr a)) (size a))
        #true)
    #false)
```

The proof is done by applying the `natural?/size` and `if-true` axioms, which allow us to rewrite this formula as

```
(if (pair? a)
    (< (size (cdr a)) (size a))
    #true)
```

and `size/cdr` allows to reduce the expression to

```
(if (pair? a)
    #true
    #true)
```

It is now easy to see that this expression is `equal?` to `#true` (by `if-same`).

5.4 Induction and recursion

A typical recursive definition consists of one or more base case, and a rule which explains how to construct/analyze a more complex object from/in terms of simpler objects.

When we want to prove that a recursive function possesses a certain property, it is sufficient to prove that it possesses that property for the simplest cases, and that the recursive transformation preserves that property.

The proofs that have this structure are called *inductive proofs* or *proofs by induction*. More specifically, an inductive proof of a claim regarding a recursive function is a proof of an *inductive claim*.

It is not always obvious what the inductive claim for a given claim should be. In general, the structure of an inductive claim should somehow correspond to the structure of recursion of one or more functions involved in that claim.

5.4.1 List induction

A particularly common case of induction is called a *list induction*. We say that an object is a `list?` either if it is `equal?` to `'()`, or if it is a `pair?` whose `cdr` is a `list?`:

```
(define (list? l)
  (or (equal? l '())
      (and (pair? l)
           (list? (cdr l)))))
```

It therefore seems natural, that an induction over list should consider `'()` as its base case, and the claim should be preserved by the `cons` or `cdr` operation.

5.4.2 An example: associativity of `append`

Consider the following theorem regarding the `append` function defined in previous section:

```
(define (associative?/append l1 l2 l3)
  (if (and (list? l1) (list? l2) (list? l3))
      (equal? (append l1 (append l2 l3))
              (append (append l1 l2) l3))))

(assure associative?/append)
```

The inductive claim

List induction over the `l1` argument provides us with the following claim:

```
(if (equal? l1 '())
    [if (and (list? l1) (list? l2) (list? l3))
        (equal? (append l1 (append l2 l3))
                (append (append l1 l2) l3))]
;else
    (if (if (and (list? l1) (list? l2) (list? l3))
            (equal? (append l1 (append l2 l3))
                    (append (append l1 l2) l3)))
        (if (and (list? (cons x l1)) (list? l2) (list? l3))
            (equal? (append (cons x l1) (append l2 l3))
                    (append (append (cons x l1) l2) l3)))))
```

The base case

It is easy to see that if we substitute `'()` for `l1` in `associative?/append` by `equal-if` in the consequent of the main `if` expression (the base case), we get

```
(if (and (list? '()) (list? l2) (list l3))
   (equal? [append '() (append l2 l3)]
           (append [append '() l2] l3)))
```

By expanding the definitions of `append` in the square brackets, we get the consequent of the main `if` expression to become `(equal? (append l2 l3) (append l2 l3))`, which proves the base case.

The inductive step

The inductive step is the alternative of the main `if` expression:

```
(if (equal? l1 '())
    #true
;else
  (if (if (and (list? l1) (list? l2) (list? l3))
          (equal? (append l1 (append l2 l3))
                  (append (append l1 l2) l3)))
      (if (and (list? (cons x l1)) (list? l2) (list? l3))
          (equal? [append (cons x l1) (append l2 l3)]
                  (append [append (cons x l1) l2] l3)))))
```

5.4.3 Axioms for `cons`

In order to prove it, we need to assert some additional axioms which establish the relationship between `cons`, `car`, `cdr` and `pair?`:

```
(define (car/cons x y)
  (equal? (car (cons x y)) x))

(assume car/cons)

(define (cdr/cons x y)
  (equal? (cdr (cons x y)) y))

(assume cdr/cons)

(define (pair?/cons x y)
  (equal? (pair? (cons x y)) #true))

(assume pair?/cons)

(define (cons/car+cdr x)
  (if (pair? x)
    (equal? x (cons (car x) (cdr x)))))

(assume cons/car+cdr)

(define (cons-equal-car x y z)
  (equal? (equal? (cons x z) (cons y z))
          (equal? x y)))

(assume cons-equal-car)

(define (cons-equal-cdr x y z)
  (equal? (equal? (cons x y) (cons x z))
          (equal? y z)))

(assume cons-equal-cdr)
```

The proof of `append` continued

Continuing our proof, we can now substitute the expressions `(append (cons x l1) ...)` with the body of the definition of `append`:

```
(if (if (and (list? l1) (list? l2) (list? l3))
        (equal? (append l1 (append l2 l3))
                (append (append l1 l2) l3)))
    (if (and (list? (cons x l1)) (list? l2) (list? l3))
        (equal? (if [pair? (cons x l1)]
                    (cons [car (cons x l1)]
                          (append [cdr (cons x l1)]
                                  (append l2 l3)))
                    ###)
                (append (if [pair? (cons x l1)]
                            (cons [car (cons x l1)]
                                  (append [cdr (cons x l1)] l2))
                            ###) l3))))
```

Some irrelevant bits of the expression were replaced with `###`. They appear twice in the alternatives of `if` expressions whose conditions are `[pair? (cons x l1)]`, which – by virtue of `pair?/cons`, are `equal?` to `#true`.

Similarly, the expressions `[car (cons x l1)]` can be replaced with `x` by `car/cons`, and `[cdr (cons x l1)]` can be replaced with `l1` by `cdr/cons`, yielding

```
(if (if (and (list? l1) (list? l2) (list? l3))
        (equal? (append l1 (append l2 l3))
                (append (append l1 l2) l3)))
    (if (and (list? (cons x l1)) (list? l2) (list? l3))
        (equal? (cons x (append l1 (append l2 l3)))
                [append (cons x (append l1 l2)) l3])))
```

By applying the same trick to the right-hand side of the innermost `equal?`, we get

```
(if (if (and (list? l1) (list? l2) (list? l3))
        (equal? (append l1 (append l2 l3))
                (append (append l1 l2) l3)))
    (if (and [list? (cons x l1)] (list? l2) (list? l3))
        (equal? (cons x (append l1 (append l2 l3)))
                (cons x (append (append l1 l2) l3)))))
```

It should be clear that in order to prove the claim, we need to appeal to the inductive hypothesis. In order to do so, we first need to unify their assumptions. By definition, `[list? (cons x l1)]` is true if `(cons x l1)` is either `'()` or if it is a `pair?` and its `cdr` is a `pair?`. Therefore, the expression reduces to `(pair? l1)`, making the conditions of inductive premise and conclusion identical.

This allows us to perform the `if-lifting` and transform the whole expression to

```
(if (and (list? l1) (list? l2) (list? l3))
    (if (equal? (append l1 (append l2 l3))
                (append (append l1 l2) l3))
        [equal? (cons x (append l1 (append l2 l3)))
                (cons x (append (append l1 l2) l3)]))
```

The application of `cons-equal-cdr` to the innermost `equal?` yields

```
(if (and (list? l1) (list? l2) (list? l3))
    (if [equal? (append l1 (append l2 l3))
                (append (append l1 l2) l3)]
        (equal? [append l1 (append l2 l3)]
                (append (append l1 l2) l3))))
```

We can now use the `equal-if` axiom to rewrite whichever side of equality we choose, say, left to right:

```
(if (and (list? l1) (list? l2) (list? l3))
    (if (equal? (append l1 (append l2 l3))
                (append (append l1 l2) l3))
        (equal? (append (append l1 l2) l3)
                (append (append l1 l2) l3))))
```

It now suffices to apply `equal-same` to the innermost `equal?` and then `if-same` two or three times to complete the inductive proof.

5.5 Conclusion

The purpose of the proofs presented in this chapter was to exemplify some methods that are useful for proving properties of programs. It is hard to deny that – without any assistance from computer tools that help to trace nested parentheses – the structures of expressions may seem obscure, and indeed, some more advanced typesetting features would certainly be helpful. We hope that the presentation was instructive nevertheless.

ACL2, the descendant of the original Boyer-Moore system, is capable of proving a large class of theorems about programs automatically using some principles that were laid out in this chapter. The source codes for ACL2 are publicly available[9].

This chapter ends the first part of this work, whose purpose was to present various tools that can be helpful for the task that we set to ourselves in the first chapter.

[9] `https://github.com/acl2/acl2`

Part II

The Substance

6

List recursion and array-receiving style

By now, we should have a fairly detailed idea how the Scheme programs ought to be executed on register machines, and how to check whether our programs posses certain properties that are of interest to us.

In this chapter we will try to formulate certain properties that should be useful for us if we wish to make our compiler use arrays in place of linked lists.

6.1 Some examples

Before we move on to the `qsort` example from the first chapter, we ought to note, that there are some much simpler examples of recursive functions whose behavior is sub-optimal.

6.1.1 The canonical implementation of `map`

Consider, for example, the following (Canonical) implementation of the `map` function, which was explained in the first chapter[1]:

```
(define (map f l)
  (if (null? l)
    '()
  ;else
    (cons (f (car l)) (map f (cdr l)))))
```

[1] For the clarity of presentation, we are not going to employ the `match` and `quasiquote` macros in our subject programs, and only use them in the meta-programs.

6.1.2 A tail-recursive variant: `reverse-map`

Despite the fact that Scheme compilers perform the tail call optimization, the above definition of `map` is not tail-recursive, so the depth of the call stack is proportional to the length of the list.

It could be rewritten to be tail-recursive in the following way:

```
(define (reverse-map f l)

  (define (traverse in out)
    (if (null? in)
      out
    ;else
      (traverse (cdr in) (cons (f (car in)) out))))

  (traverse l '()))
```

The problem with this function is that that elements of the output list are in the reverse order – for example, `(reverse-map square '(1 2 3))` would construct a list `(9 4 1)`. Of course, we could now define `map` by using `reverse-map` with the identity function

```
(define (map f l)
  (reverse-map f (reverse-map (lambda (x) x) l)))
```

and while, defined this way, `map` would indeed use a constant amount of stack space, it would traverse the list twice, and needlessly generate `(length l)` cons-cells of garbage.

6.1.3 A destructive variant: `map!`

Another problem is that if the list passed to `map` isn't going to be used any more, the storage allocated for this list will be unavailable until the next cycle of garbage collection, and its reclaim will occupy some running time.

This problem could be solved by the destructive version of `map`:

```
(define (map! f l)
  (define (iterate point)
    (if (null? point)
      l
    ;else
      (begin
        (set-car! point (f (car point)))
        (iterate (cdr point)))))
  (iterate l))
```

This variant is both tail-recursive and economical, but it only works if the aforementioned condition is satisfied, i.e. the original list isn't used anywhere else in the program.

Still, it could be reasonable (under some circumstances) to have our compiler detect the conditions like this, and replace some references to `map` with references to `map!`.

6.2 Array passing

Lastly, we noted in the first chapter, that linked lists, although conceptually simple and pragmatically versatile, are not always the most fortunate structure to store the data, since the computer memories are typically organized as arrays.

We therefore devise a systematic procedure for transforming some class of functions that operate on lists into functions operating on arrays (or directly on memory pointers).

A function should receive an additional argument, `target`, which should be used for storing the result. For example, the `map` function should become:

```
(define (map-into target f l)

  (define (step target-index f l-index)
    (if (beyond? l-index l)
      target
      (begin
        (memory-set! target-index (f (memory-ref l-index)))
        (step (next target-index) f (next l-index)))))

  (step (start target) f (start l)))
```

The above code refers to some procedures that are a part of our interface to arrays, namely `beyond?`, `next`, `start`, `memory-ref` and `memory-set!`.

The `start` function returns a pointer to the first element of array, the `beyond?` predicate checks whether a given pointer points outside of an array, and `next` function returns a pointer of an element next to a given one. The `memory-ref` and `memory-set!` procedures return and modify the value of memory cell pointed to by a given pointer.

For now, we are deliberately avoiding to provide any concrete implementation of this interface, so that we don't need to decide whether all the sizes of elements of an array are uniform, nor whether the addresses of subsequent elements should be ascending or descending.

The questions that arise are:

1. Under what circumstances can we transform a recursive function to the array-receiving style?

2. How do we transform a recursive function to the array-receiving style?

3. How do we transform regular function calls into array-passing function calls?

We don't know the exact answers to these questions, but we shall propose an initial attempt of addressing them.

6.2.1 List recursion

Dubbing the term *tail recursion*, we shall name the circumstances under which we can use arrays instead of lists, a *list recursion* (to be distinguished from *tree recursion* or *free recursion*).

The first approximation of a list recursion is that it is a function `f` whose tail expression is either:

- a list literal (or a call to a function which produces a list literal), or
- an argument to `f` which is known to be bound to a list (either by virtue of some assertion, or proven within the `f`'s calling context), or
- a recursive call to `f`, or
- an expression of the form `(cons x (f . args))`, where neither `x` nor `args` contains a recursive call to `f`.

The last condition could actually be loosened a bit: the second argument to `cons` could itself be a `cons`, and so on, until we make the recursive call, or it could be a call to a function which evaluates to a list or to an expression of the form `(cons x y)`, where `y` is an argument of that function, and it is bound to the value of the recursive call `(f . args)`. However important, these nuances obscure the point that we are trying to make, so we shall ignore them for the moment.

In order to put what we have just said more formally, we need to specify what we mean by tail expressions of a given expression. When our expression has a form `(if <test> <then> <else>)`, then its tail expressions are the tail expressions of `<then>` and tail expressions of `<else>`. Otherwise, if it has the form `((lambda <args> <body>) . <values>)`, then the tail expressions are the tail expressions of `<body>` with `<values>` substituted for `<args>` throughout. Otherwise, the tail expressions are a singleton containing only the expression itself:

```
(define (tail-expressions expression)
  (match expression
    (('if <test> <then> <else>)
     `(,@(tail-expressions <then>) ,@(tail-expressions <else>)))
    ((('lambda <args> <body>) . <values>)
     (tail-expressions (substitute <args> <values> <body>)))
    (_
     `(,expression))))
```

To check whether a function is list-recursive, we need to know its name, its body and a list of its arguments. The name is needed, because the function is potentially recursive, and if so, it refers to itself by that very name. The arguments are needed, because it is possible that a function returns some of its arguments in its tail position. The body is needed for the obvious reasons.

```
(define (list-recursion? name+args+body)
  (let* (((name args body) name+args+body)
         (tail-expressions (tail-expressions body)))
    (define (list-recursive? tail)
      (match tail
        (('quote literal)
         (list? literal))

        (('cons item (function . arguments))
         (and (not (calling? item name))
              (equal? function name)
              (every (lambda (arg)
                       (not (calling? arg name)))
                     arguments)))

        ((('lambda <args> <body>) . <values>)
         (list-recursive? (substitute <args> <values> <body>)))

        ((function . arguments)
         (and (every (lambda (arg)
                       (not (calling? arg name)))
                     arguments)
              (equal? name function)))

        (_
         (is tail member args))))

    (every list-recursive? tail-expressions)))
```

where the `calling?` predicate is defined as:

```
(define (calling? expression name)
  (match expression
    (('quote _)
     #false)

    ((('lambda <args> <body>) . <values>)
     (calling? (substitute <args> <values> <body>) name))

    ((function . args)
     (or (equal? function name)
         (calling? function name)
         (any (lambda (arg)
                (calling? arg name))
              args)))
    (_
     #false)))
```

While these notions clearly need an elaboration (for example, the fact that we use `substitute` with recursive call allows to construct forms that would never terminate), they should be sufficient to indicate some conditions that permit us to convert a list recursive function to the array-receiving style.

6.2.2 Transformation to array-receiving style

The `array-receiving` function takes a triple `(name args body)` and returns a definition of an array-receiving version of that function.

For example (as explained earlier),

```
(array-receiving '(map (f l)
                       (if (null? l)
                         '()
                         (cons (f (car l))
                               (map f (cdr l))))))
```

should return something like the following definition:

```
(define (map-into target f l)

  (define (step target-index f l-index)
    (if (beyond? l-index l)
      target
      (begin
        (memory-set! target-index (f (memory-ref l-index)))
        (step (next target-index) f (next l-index)))))
```

```
(step (start target) f (start l)))
```

Clearly, a list recursive function is transformed into a helper function called `step`, which is invoked from the body of the array-receiving function (with the `-into` suffix with its name and an additional parameter called `target`).

The parameters of the `step` helper function are just like the parameters of the array-receiving function, except that some of its arguments which originally referred to lists, now become pointers to arrays (which is signified with tagging them with the `-index` suffix).

Note that in general it could be the case that some arguments that referred to lists in the original function would still refer to lists, rather than arrays: we may want to only capture the result to the array, and we may not necessarily wish to have to convert every input list to an array prior to the function call. We could therefore assure the fine-grained control by explicitly passing the arguments that should be interpreted as arrays in the array-receiving version of a function.

However, for the purpose of this work, we assume that all the arguments that can be inferred to refer to lists shall be treated as arrays.

We can infer that an argument refers to a list, if it is tested to be `null?`, or if either its `car` or `cdr` is applied to it. (Again, this heuristics may not be particularly comprehensive, but it should be sufficient for the purpose of this work.)

```
(define (list-arguments expression)

  (define (possibly argument)
    (cond ((symbol? argument)
           `(,argument))
          ((pair? argument)
           (list-arguments argument))
          (else
           '())))

  (match expression
    ((function . args)
     (if (is function member '(cdr null?))
         (apply union (map possibly args))
         (apply union (map list-arguments args))))
    (_
     '())))
```

As in the case of the `list-recursion?` test, the function that converts a function to the `array-receiving` style should take a triple containing name, arguments and body:

```
(define (array-receiving name+args+body)
  (let* (((name args body) name+args+body)
         (list-args (intersection args (list-arguments body))))
```

The conversion needs to consider a few cases: branching instruction should convert both branches recursively; a `cons` instruction should be converted to a write of its first argument into the `target-index`, followed by the conversion of the second argument. A `null?` test should be converted to a call to `beyond?`. A recursive call in original function should be transformed into a recursive call to `step`, where all the applications of `cdr` to any of `list-args` should be replaced with calls to `next`, and applications of `car` – to calls to `memory-ref`. In other words, we need at least two helper functions – `convert`, which converts the whole expressions, and `convert-argument` to convert arguments to function applications:

```
(define (convert-argument argument)
  (match argument
    (('car expression)
     (if (is expression member list-args)
         `(memory-ref ,(symbol-append expression '-index))
     ;else
          argument))

    (('cdr expression)
     (if (is expression member list-args)
         `(next ,(symbol-append expression '-index))
     ;else
          argument))

    ((function . args)
     `(,function . ,(map convert-argument args)))
    (_
     argument)))

(define (convert expression next-target)
  (match expression
    (('if <test> <then> <else>)
     `(if ,(convert <test> next-target)
          ,(convert <then> next-target)
          ,(convert <else> next-target)))

    (('cons first rest)
     `(begin
        (memory-set! target-index ,(convert-argument first))
        ,(convert rest `(next ,next-target))))
```

```
    (('null? x)
     (if (is x member list-args)
         `(beyond? ,(symbol-append x '-index) ,x)
     ;else
         expression))

    (('quote ())
     'target)

    ((function . arguments)
     (if (eq? function name)
        `(step ,next-target
               . ,(map convert-argument arguments))
     ;else
        `(,function . ,(map convert-argument arguments))))

    (_
     (if (is expression member list-arguments)
         (symbol-append expression '-index)
     ;else
         expression))))
```

The `array-receiving` function needs to construct a `define` form containing a definition of the `step` function and an invocation of that function with initial arguments:

```
`(define (,(symbol-append name '-into) target . ,args)
   (define (step target-index
                 . ,(map (lambda (arg)
                           (if (is arg member list-args)
                               (symbol-append arg '-index)
                           ;else
                               arg))
                         args))
     ,(convert body 'target-index))

   (step (start target)
         . ,(map (lambda (arg)
                   (if (is arg member list-args)
                      `(start ,arg)
                   ;else
                      arg))
                 args)))))
   ;; the body of `array-receiving ends here
```

Of course, there are means which allow to avoid accidental name clashes with the names such as `step` or `target`, but for the time being we ignore this issue completely.

While the `array-receiving` function may not be perfect, it is general enough to be able to transform some functions other than `map`. For example, one can easily check that the array-receiving version of the function `range` defined as

```
(define (range lo hi)
  (if (> lo hi)
      '()
  ;else
      (cons lo (range (+ lo 1) hi))))
```

is the following:

```
(define (range-into target lo hi)
  (define (step target-index lo hi)
    (if (>= lo hi)
        target
    ;else
        (begin
          (memory-set! target-index lo)
          (step (next target-index) (+ lo 1) hi))))
  (step (start target) lo hi))
```

Likewise, the canonical implementation of the `filter` function (as explained in chapter 2)

```
(define (filter p l)
  (if (null? l)
      '()
  ;else
      (if (p (car l))
          (cons (car l) (filter p (cdr l)))
      ;else
          (filter p (cdr l)))))
```

and its corresponding array-receiving version is

```
(define (filter-into target p l)
  (define (step target-index p l-index)
    (if (beyond? l-index l)
      target
    ;else
      (if (p (memory-ref l-index))
        (begin
          (memory-set! target-index (memory-ref l-index))
          (step (next target-index) p (next l-index)))
      ;else
        (step target-index p (next l-index)))))
  (step (start target) p (start l)))
```

6.2.3 Memory management

The universality of Lisp based systems stems from the fact the `cons` operator, invoked from within functions, is responsible for memory allocation, and the responsibility for reclaiming the memory that is no longer in use belongs to the garbage collector.

The array receiving style, however, transfers the burden of memory allocation from a callee to a caller. In order for this to be possible, the caller needs to know how much memory the called function is going to need, and allocate it prior to the call, or – if it is able to prove that some sufficiently large area of memory won't be used in the rest of the program – reuse some previously allocated area.

While this problem can be hard to determine in general, there are clearly situations when it is relatively easy. For example, it should not be hard to prove the following lemmas (assuming totality of `f` and `p`):

```
(define (map-length f l)
  (if (and (list? l) (unary-function? f))
    (equal? (length (map f l))
            (length l))))

(assure map-length)

(define (append-length a b)
  (if (and (list? a) (list? b))
    (equal? (length (append a b))
            (+ (length a) (length b)))))

(assure append-length)
```

```
(define (range-length lo hi)
  (if (and (natural? lo) (natural? hi))
    (equal? (length (range lo hi))
            (max 0 (- hi lo)))))

(assure range-length)

(define (filter-length p l)
  (if (and (list? l) (unary-predicate? p))
    (equal? (max (length l) (length (filter p l)))
            (length l))))

(assure filter-length)
```

These lemmas can be used to infer the amount of memory that needs to be allocated for a given function. Note that, depending on situation, this information doesn't necessarily need to be available prior to a function call: one can imagine that the `target` argument to an array-receiving function could be located at the end of the heap, and grow the array as needed.

Reusing memory

However, if we are able to infer the size of the output of a function prior to the call, we could potentially overwrite some object which would no longer be needed for the computation. This in turn could decrease program's reliance on garbage collection, increasing overall performance.

This observation, in turn, leaves us with the following question: how can we know the lifetimes of heap allocated objects? The intuitive answer is that these lifetimes span between the creation of an object, and the last point at which any of the variables referring to that object is used.

One can imagine at least two counterexamples to this intuition, though. The first one is a function which might return some of its arguments. Consider the following procedure:

```
(define (random-argument . arguments)
  (list-ref arguments (random (length arguments))))
```

Unless we make a function like this make a copy of its return value (which would likely be unreasonable, given the goal we set to ourselves), we cannot rely on the fact that any of its arguments is no longer used in the code following the call to that function, at least as long as the result of that procedure is used thereby.

Of course, it might be tempting to ask, under which circumstances can we prove that a function does not return any data structure that is shared by any of its arguments, and this is indeed an interesting question. For the time being, we shall allow ourselves to leave it unanswered, though.

Shared objects

The second example is of a greater significance to us, because it has more to do with the goal that we set to ourselves in the first chapter. It might be the case (often a desirable one), that a list produced by some function should be a part of another list. In particular, if a function application (or its result) is the first argument to `append`, we would wish to arrange the computation just by placing the memory areas of its arguments side by side, avoiding any actual calls to `append` and memory copying whatsoever.

It therefore seems that the question, how do we prove that an allocated object is no longer needed, is in general non-trivial, and instead of solving it for the general case, we need to focus on some particular cases that serve our goal.

Let's consider a simple example of the aforementioned optimization of `append`:

```
(define (numbers&squares amount)
  (let* ((numbers (range 0 amount))
         (squares (map square numbers)))
    (append numbers squares)))
```

We can infer from the `range-length` lemma, that the length of `numbers` is `amount`, and likewise – from `map-length` – that the length of `squares` is the length of `numbers`, i.e. `amount`. Finally, by `append-length` we could conclude that the amount of memory that has to be allocated for `numbers&squares` is `(+ amount amount)`.

We could therefore expect that – if the `numbers&squares` function isn't itself meant to be array-receiving, it could be transformed to the following form:

```
(define (numbers&squares/array amount)
  (let* ((memory (allocate (+ amount amount)))
         (numbers (range-into (view memory 0 amount)
                              0 amount))
         (squares (map-into (view memory amount amount)
                            square numbers)))
    memory))
```

How can this transformation be performed? Speaking most generally, we need to transform it from the last expression to the first. We observe, that the tail expression in the original function is `(append numbers squares)`. Subsequently, we notice that `numbers` and `squares` are the results of functions that are list-recursive, and – according to the argument presented earlier – we are able to calculate their lengths and – consequently – the total amount of memory that needs to be allocated by the function.

Because `append` prompts us to allocate a single block of memory to hold the results of both `range` and `map`, we need to be able to access some *fragments* of the allocated area. In order to do so, we can use what is traditionally called *views* of memory, which – in the system we're designing – can be created using the `view` function. As in the case of other functions relating to arrays, its implementation would depend on the particular representation of the meta-data for arrays.

6.2.4 Explicit allocation

In chapter 4, while designing our compiler, we perceived functions as black boxes, or abstraction barriers, that allowed us to treat each component of the program separately.

It should be apparent that, since chapter 5, we have been looking at functions as *white boxes*: we knew every detail about the functions we were dealing with, which allowed us to draw interesting conclusions about them.

Here, we shall assume that our program consists of definitions only. Some of these definitions will be list-recursive, and thus shall be converted to array-receiving style. We also expect that there are some functions that call these array-receiving ones. They shall be responsible for allocating the memory needed by those array-receiving ones, and thus we shall call them "array-passing" (or `senders`). Lastly, we expect that there are functions that neither pass nor receive arrays. They shall remain intact.

In order to simplify the reasoning about array-passing functions, we shall perform a complete β reduction of their bodies, that is – all substitutions of applications of the λ expressions used in the bodies of their definitions. For example, if we expand the `let*` form of `numbers&squares`, we obtain

```
(define (numbers&squares amount)
  ((lambda (numbers)
     ((lambda (squares)
        (append numbers squares))
      (map square numbers)))
   (range 0 amount)))
```

which contains two applications of `lambda` forms. We can reduce this expression to

```
(define (numbers&squares amount)
  (append (range 0 amount) (map square (range 0 amount))))
```

The reduction can be defined easily using our substitution function:

```
(define (reduce expression)
  (match expression
    (('quote _)
     expression)
```

```
((’if <test> <then> <else>)
 `(if ,(reduce <test>)
      ,(reduce <then>)
      ,(reduce <else>)))

(((’lambda <args> <body>) . <values>)
 (reduce (substitute <args> <values> <body>)))

((operator . operands)
 `(,(reduce operator) . ,(map reduce operands)))

(_
 expression)))
```

As we can see, a sequence of operations becomes a complex expression (in a way, this transformation is an opposite of the CPS conversion from chapter 4, which transformed complex expressions into step by step computations).

It may seem displeasing that the `(range 0 amount)` application is repeated twice in the resulting expression. However, our transformation could eliminate repeating applications rather easily.

In order to estimate the amount of memory needed for the result of an array-receiving function, we need to synthesize the knowledge from our lemmas. Of course, this is something that a system should do for us, but devising a method for synthesizing the `size`[2] function for a particular set of array-receiving functions is beyond the scope of this work. Here we assume that we know how to compute the size for `map`, `filter` and `range`:

```
(define (size expression)
  (match expression
    ((’if <test> <then> <else>)
     `(max ,(size <then>) ,(size <else>)))

    (((’lambda <args> <body>) . <values>)
     (size (substitute <args> <values> <body>)))

    ((’map f l)
     (size l))

    ((’range lo hi)
     `(max 0 (- ,hi ,lo)))
```

[2] We use the name `size` in a sense that is different than was characterized in chapter 5, where it meant the number of cons cells used by an object. Here we desire the `size` to mean the number of elements of a sequence. We hope that the reader doesn't get confused with this ambiguity.

```
    (('filter p l)
     (size l))

    (('append x y)
     `(+ ,(size x) ,(size y)))

    (('cons x y)
     `(+ 1 ,(size y)))

    (_
     `(length ,expression))))
```

The transformation itself needs to split the definitions into the categories specified at the beginning of this section:

```
(define (array-passing-library definitions)
  (let* ((((’define (names . args) bodies) ...) definitions)
         (functions (zip names args bodies))
         (list-recursive non-list-recursive
                         (partition list-recursion? functions))
         (senders intact
                  (partition (lambda ((name args body))
                               (let ((called (called-functions
                                              `(lambda ,args
                                                 ,body))))
                                 (any (lambda ((receiver _ _))
                                        (is receiver member called))
                                      list-recursive)))
                             non-list-recursive)))
```

It will turn out, that we will want to check whether a given expression is an application of an array-receiving function, i.e. whether it belongs to the `list-recursive` set[3]:

```
(define (array-receiving? expression)
  (and-let* (((function . _) expression))
    (any (lambda ((name _ _))
           (eq? name function))
         list-recursive)))
```

The array-passing versions of the functions should, prior to the call, estimate the amount of memory that can be used by the called functions, and allocate storage for the results. Then it should create the views for each invocation of array-receiving function, and pass them as suitable.

[3]For brevity, we've decided to use the `and-let*` special form, inspired by the SRFI-2 document[52], which was extended to support pattern matching.

```
(define (array-passing name+args+body)
  (let* (((name args body) name+args+body)
         (expression (reduce body)))
    `(define (,name . ,args)
       (let ((memory (allocate ,(size expression))))
         ,@(pass-result expression 0)
         memory))))
```

The actual transformation of a function to the array-passing form is much harder, and – as before – we do not dare to claim that the transformation is correct or complete (other than that it works for some cases that were tested by us).

For the time being, we assume that the main expression of our function is either a call to `append` (which is treated specially), whose both arguments are the calls to some array receiving functions, or the main expression itself is a call to some array-receiving function, which can further contain some calls to other array-receiving functions in its arguments.

The case of `append` is rather straightforward[4] – we just need to pass the results of the nested calls to subsequent fragments of the allocated array:

```
(define (pass-result expression base-address)
  (match expression
    (('append x y)
     `(,@(pass-result x base-address)
       ,@(pass-result y `(+ ,base-address ,(size x)))))))
```

The invocation of an `array-receiving?` function is much more complicated: we need to see whether the call itself contains any calls to array-receiving functions in the positions of `list-arguments`, and if so, we need to convert these calls as well.

```
((function . arguments)
 (let* (((name args body) (find (lambda ((name _ _))
                                  (equal? name function))
                                list-recursive))
        (list-args (intersection (list-arguments body)
                                 args))
        (args/array-passing (filter array-receiving?
                                    arguments)))
```

It may not be immediately obvious, but the `list-args` is bound to a list of formal parameters of the function being called, which are known to

[4] To an extent: note that while the method that we chose allows to treat functions whose exact output size can be inferred in advance, it fails for the cases like `filter`, where we can only know the maximum size of the output before actually running the function.

be bound to lists, whereas `args/array-passing` refers to a list of actual values being passed to the given function. (This ambiguity in terminology seems to be a recurring topic in this chapter, actually, and we are sorry for not having done anything about it.)

We assume that the elements in these lists correspond to each other – that each of the `list-arguments` corresponds to the invocation of an array-passing function.

We are also making a few other nasty assumption here, namely – that the amount of memory used to store the return values of functions called from argument positions in some other function application does not exceed the amount of memory allocated for the main function's result, and that the function main can overwrite these results as it goes.

Note, that although there are occasions where this actually is the case (such as the `map` function), it would not be difficult to construct a counterexample.

Nevertheless, we do not allow two functions called from within the same level of nesting to overwrite the result of one another. In order to do so, we construct a sequence of "base addresses" of the allocated area of memory, where a function is allowed to write.

We decided to have these base addresses grow from left to right:

```
(define (argument-bases initial-base parameters values)
  (match `(,parameters ,values)
    ((() ())
     '())
    (((parameter . parameters) (value . values))
     (or (and-let* (((is parameter member list-args))
                    ((array-receiving? value))
                    (base `(+ ,initial-base
                              ,(size value))))
           `(,initial-base . ,(argument-bases
                               base
                               parameters
                               values)))
         (argument-bases initial-base
                         parameters
                         values)))))
```

The sequence of base addresses constructed in this way can be used to generate the calls to the `view` function, which allows to access a particular fragment of the allocated memory:

```
(define (argument-views arguments bases)
  (match arguments
    (()
     '())
    ((argument . arguments)
     (if (array-receiving? argument)
         (let* (((base . bases) bases))
           `((view memory ,base ,(size argument))
             . ,(argument-views arguments bases)))
         ;else
         `(,argument . ,(argument-views
                         arguments bases))))))
```

With these auxiliary definitions at hand, we can express the transformation to array passing style, which consists of two parts: the invocation of a given function should be preceded with invocations of all array-receiving arguments, and all the references to the results of those functions within the invocation of the main function should be replaced with references to the appropriate memory fragments (or `view`s):

```
(let ((bases (argument-bases base-address
                             args
                             arguments))
      (pass-into (symbol-append function '-into)))
  `(,@(append-map pass-result
                  args/array-passing
                  bases)
    (,pass-into (view memory
                      ,base-address
                      ,(size expression))
                . ,(argument-views arguments bases)))))))
;; the definition of ``pass-result'' ends here
```

The main expression of the `array-passing-library` simply converts all the `list-recursive` definitions to the `array-receiving` form, and every caller of those functions – to the `array-passing` form:

```
`(,@(map (lambda ((name args body))
           `(define (,name . ,args) ,body))
         intact)
  ,@(map array-receiving list-recursive)
  ,@(map array-passing senders))))
;; the definition of ``array-passing-library'' ends here
```

6.2.5 Examples revisited

We can check that the function actually works. The invocation of

```
(array-passing-library
 '((define (map f l)
     (if (null? l)
         '()
         (cons (f (car l))
               (map f (cdr l)))))

   (define (square x)
     (* x x))

   (define (range lo hi)
     (if (> lo hi)
         '()
         (cons lo (range (+ lo 1) hi))))

   (define (numbers&squares amount)
      ((lambda (numbers)
         ((lambda (squares)
            (append numbers squares))
          (map square numbers)))
       (range 0 amount)))))
```

produces the following result:

```
((define (square x)
   (* x x))

 (define (map-into target f l)
   (define (step target-index f l-index)
     (if (beyond? l-index l)
       target
       (begin
         (memory-set!
           target-index
           (f (memory-ref l-index)))
         (step (next target-index) f (next l-index)))))
   (step (start target) f (start l)))
```

```
(define (range-into target lo hi)
  (define (step target-index lo hi)
    (if (> lo hi)
      target
      (begin
        (memory-set! target-index lo)
        (step (next target-index) (+ lo 1) hi))))
  (step (start target) lo hi))

(define (numbers&squares amount)
  (let ((memory (allocate (+ (max 0 (- amount 0))
                             (max 0 (- amount 0))))))
    (range-into (view memory 0 (max 0 (- amount 0)))
                0 amount)
    (range-into (view memory (+ 0 (max 0 (- amount 0)))
                      (max 0 (- amount 0)))
                0 amount)
    (map-into (view memory (+ 0 (max 0 (- amount 0)))
                    (max 0 (- amount 0)))
              square
              (view memory (+ 0 (max 0 (- amount 0)))
                    (max 0 (- amount 0))))
    memory)))
```

Note that the definition of `numbers&squares` differs significantly from what we have anticipated earlier (cf. the definition of `numbers&squares-/array` on page 103).

The most apparent difference is that the arguments to the `view` function are terribly cluttered. Assuming that `max` is non-negative, we could transform the above definition of `numbers&squares` obtained from `array--passing-library` into:

```
(define (numbers&squares amount)
  (let ((memory (allocate (+ amount amount))))
    (range-into (view memory 0 amount) 0 amount)
    (range-into (view memory amount amount) 0 amount)
    (map-into (view memory amount amount)
              square
              (view memory amount amount))
    memory))
```

Now the most significant difference is that the `range-into` function is invoked twice, and that the results from `map-into` and `range-into` are never captured or passed explicitly.

The first difference could be fixed by elaborating our method for obtaining `array-passing` versions of functions: after performing the `reduce` operation, we would need to extract common sub-expressions and perform some analysis to see whether some calls could be eliminated without any harm to the result of the computation.

The fact that arguments aren't passed explicitly, but only cause side effects on the content of the `memory` area, is more bothersome in the case of functions such as `filter`, whose result size cannot be known *a priori*.

6.3 Conclusion

We have presented a sketch of a method which allows us to transform functional programs that operate on lists with destructive programs that operate on arrays.

Surely, the method is far from perfect, but it works for some simple examples.

Some problems, like the synthesis of the `size` function (defined on page 105) for a given set of array-receiving definitions, require some elaboration and seem to form whole research topics on their own.

Another ones – like the lifetime analysis of some specific heap areas – reveal a lot of similarities to the topics that are already well examined in the Computer Science, and in the field of compiler construction in particular.

7

Trying to make Quicksort quick again

7.1 Motivating examples revisited

In the first chapter we have shown an implementation of `qsort` function in Haskell, contrasted with an imperative implementation of Tony Hoare's QUICKSORT algorithm. We claimed that although the Haskell program possessed certain properties that are desired from the point of view of systematic program construction, the ease of its analysis and compositionality came with a cost, namely – an increased consumption of computational resources.

We further claimed that this cost can be reduced – even to zero – in the presence of appropriate reasoning and transformation tools, and then went on to presenting some tools that could be helpful in achieving that goal.

In this chapter, we are going to attempt to apply these tools to our original problem. In order to do so, however, we first need to translate the programs to languages that are more susceptible to meta-programming than Haskell or pseudo-code.

7.1.1 Implementing `qsort` in Scheme

The implementation of `qsort` in Scheme using `quasiquote` and `match` is very similar to the Haskell version:

```
(define (qsort list)
  (match list
    (() '())
    ((head . tail)
     (let ((below (filter (is _ < head) tail))
           (above (filter (is _ >= head) tail)))
       `(,@(qsort above) ,head ,@(qsort below))))))
```

The `match` macro could be expanded to an `if` form, and the `quasiquote` macro is just a call to the `append` function in disguise:

```
(define (qsort list)
  (if (null? list)
    '()
  ;else
    (let ((head (car list))
          (tail (cdr list)))
      (let ((below (filter (lambda (x) (< x head)) tail))
            (above (filter (lambda (x) (>= x head)) tail)))
        (append (qsort above)
                (cons head (qsort below)))))))
```

7.1.2 The desired outcome

For the purpose of reference, we may write a counterpart of the QUICKSORT and PARTITION programs from chapter 1 in our assembly language:

```
Quicksort:
   (if first >= last goto end:)
   (push return)
   (return <- partitioned:)
   (goto Partition:)
partitioned:
   (push last)
   (push result)
   (return <- left-sorted:)
   (last <- result - 1)
   (goto Quicksort:)
left-sorted:
   (pop result)
   (pop last)
   (pop return)
   (first <- result + 1)
   (goto Quicksort:)
end:
   (goto return)
```

```
Partition:
   (pivot <- [last])
   (trail <- first - 1)
   (front <- first)
loop:
   (if front >= last goto done:)
   (item <- [front])
   (if item > pivot goto next:)
   (trail <- trail + 1)
   (swap <- [trail])
   ([front] <- swap)
   ([trail] <- item)
next:
   (front <- front + 1)
   (goto loop:)
done:
   (result <- trail + 1)
   (item <- [result])
   (swap <- [last])
   ([result] <- swap)
   ([last] <- item)
   (goto return)
```

It is no surprise that the assembly corresponding to QUICKSORT and PARTITION is even more difficult to follow that the original versions of those

functions. For this reason, they won't be very useful to us. Instead, we prefer to obtain an imperative version of Quicksort in Scheme.

7.1.3 Hoare partitioning

The PARTITION function presented in chapter 1 is itself rather difficult to follow, as it operates on array indices. We can achieve a significant improvement in readability by merely elaborating the interfaces to data types: rather than raw indices, we would prefer to use *array slices* (which resemble the concept of *memory views* from chapter 6).

For the remainder of this section we shall assume, that there is a `slice` function available, which takes an array, base index and size, and creates an array view that shares the storage with an original array.

For example, if `A` is defined as a Scheme array `#(1 2 3 4 5)`, then `(slice A 1 3)` shall create an array `#(2 3 4)` (let's call it `B`).

If we modify this new array, the corresponding elements of the old array will get modified as well. For example, `(array-set! B 'X 0)` will cause `B` to become `#(X 3 4)`, but will also cause `A` to become `#(1 X 3 4 5)`[1].

Given such operation, one can express the Hoare partitioning scheme in the following way:

```
(define (Hoare-partition! array)
  (let ((size (array-length array))
        (pivot (array-ref array 0)))

    (define (parts! back front)
      (cond ((is front >= size)
             (let ((middle (- back 1)))
               (swap! 0 middle array)
               (values (slice array 0 middle)
                       (slice array (+ middle 1)
                                    (- front back)))))

            ((is (array-ref array front) < pivot)
             (swap! back front array)
             (parts! (+ back 1) (+ front 1)))

            (else
             (parts! back (+ front 1)))))

    (parts! 1 1)))
```

[1] It may seem surprising that `array-set!` takes the value as its second argument, and array index as its last argument – contrary to `vector-set!` known from Scheme. The code that we're showing here has been written and tested with Guile Scheme, which supports its own API for shared and multidimensional arrays [48].

Even though it is imperative, we believe that operating on array slices rather than raw array indices provide a significant improvement over the variant presented in chapter 1.

Given this definition, the Scheme version of the QUICKSORT procedure can be defined as:

```
(define (quicksort! array)
  (unless (is (array-length array) <= 1)
    (let ((smaller greater (Hoare-partition! array)))
      (quicksort! smaller)
      (quicksort! greater)))
  array)
```

Again, it is slightly easier to follow, because the reader doesn't need to be concerned with array indexing, although in this case the difference is rather negligible.

The problem is, though, that – contrary to the `qsort` variants in Scheme and Haskell – the form of the `quicksort!` function doesn't make the idea behind quicksort immediately apparent. In particular, its correctness is based on the fact that the pivot is swapped with the "middle" element before returning slices of the array.

7.1.4 Functional variant of Hoare partitioning

The Hoare partition scheme can actually be expressed in a purely functional fashion. We generalize it a bit to take an arbitrary condition for partitioning the array:

```
(define (Hoare-partition condition list)

  (define (parts list back front)
    (cond ((is front >= (length list))
           (split-at list back))

          ((condition (list-ref list front))
           (parts (swap back front list) (+ back 1)
                                         (+ front 1)))
          (else
           (parts list back (+ front 1)))))

  (parts list 0 0))
```

where `swap` is defined as

```
(define (swap i-th #;with j-th #;in list)
  (alter i-th (alter j-th list (list-ref list i-th))
         (list-ref list j-th)))
```

and the helper `alter` function – as

```
(define (alter n-th #;in list #;with replacement)
  (let (((head . tail) list))
    (if (= n-th 0)
        `(,replacement . ,tail)
    ;else
        `(,head . ,(alter (- n-th 1) tail
                          replacement)))))
```

The `(split-at list n)` function defined in [67] returns two values – the first one contains the first `n` elements of the original `list`, and the second one – the remaining elements.

This allows us to express `qsort` in the following way:

```
(define (quicksort list)
  (match list
    (()
     '())
    ((head . tail)
     (let* ((below above (Hoare-partition (is _ < head) tail)))
       `(,@(quicksort below) ,head ,@(quicksort above))))))
```

7.2 Transformation

The question which now arises is: under what circumstances are we allowed to rewrite `quicksort` to something like `quicksort!`, and `Hoare-partition` to something like `Hoare-partition!`?

Before attempting to answer it, let's note that there are a few bothering things about the definitions of `quicksort!` and `Hoare-partition!`. As we noted before, the base case of `Hoare-partition!` contains the `(swap! 0 middle array)` instruction, which silently inserts a pivoting element of the array in between the slices.

Furthermore, the algorithm works, because the slices returned by `Hoare--partition!` belong to a continuous region.

Lastly, the `quicksort!` procedure doesn't give a clue about the structure of the result, contrary to its functional counterpart.

This observation prompts us with the following hint: perhaps we could use the structural information contained in the `` `(,@(quicksort below) ,head ,@(quicksort above)) `` expression to automatically generate a call to `swap!`?

7.2.1 Some properties of `quicksort`

`quicksort` is order-invariant

We know that this is acceptable, because `quicksort` returns the same result for any permutation of its input[2], and therefore we can permute the elements of the `below` and `above` lists anyway we like (as long as this happens before their elements are sorted).

We could express this property in the following way:

```
(define ((ordering? function) elements)
  (if (list? elements)
    (let ((result (function elements)))
      (every (lambda (permutation)
               (equal? (function permutation) result))
             (permutations elements)))))

(assure (ordering? quicksort))
```

where `permutations` can be defined using a helper function `insertions`:

```
(define (insertions x l)
  (match l
    (()
     `((,x)))
    ((head . tail)
     `((,x ,head . ,tail) . ,(map (lambda (y)
                                    `(,head . ,y))
                                  (insertions x tail))))))

(e.g. (insertions 'a '(x y z))
   ===> ((a x y z) (x a y z) (x y a z) (x y z a)))

(define (permutations l)
  (match l
    (()
     '(()))
    ((head . tail)
     (append-map (lambda (sub)
                   (insertions head sub))
                 (permutations tail)))))

(e.g. (permutations '(a b c))
   ===> ((a b c) (b a c) (b c a) (a c b) (c a b) (c b a)))
```

[2]This is only true if the `<` relation is a total order with regard to the elements contained in the input list.

The problem with the `ordering?` lemma is that it does not conform to the specification of `<rule>` from chapter 5, and hence we wouldn't know how to use it. It can be expressed in a more operational form:

```
(define ((ordering*? function) elements probe)
  (if (and (list? elements)
           (is probe member (permutations elements)))
      (equal? (function probe) (function elements))))

(assure (ordering*? quicksort))
```

`quicksort` preserves elements

```
(define ((preserves-elements? function) input probe)
  (equal? (not (is probe member input))
          (not (is probe member (function input)))))

(assure (preserves-elements? quicksort))
```

The arguments to `equal?` in `preserves-elements?` are negated, because `member` returns some truth-ish value (more precisely, it returns the suffix of its second argument, whose `car` is `equal?` to its first argument [66]), rather than just `#true`.

`quicksort` preserves length

```
(define ((preserves-length? function) input)
  (equal? (length input) (length (function input))))

(assure (preserves-length? quicksort))
```

7.2.2 Analysis continued

Consider the expanded version of the main expression from `quicksort`:

```
(append (quicksort below) (cons head (quicksort above)))
```

We assume that `(quicksort above)` is already in the right place. Therefore, we're concerned with the expression of the form

```
(append (quicksort below) (cons head DO-NOT-MOVE!))
```

We can deduce that there is a free cell available to the left of the `below` list (actually, it is occupied by the value of `head`, but we don't care about it too much, since we already managed to store this value in a local variable) and that – in order to make the result of the `append` function fit the allocated

storage – we need to move its last element into that free cell and then place the value of `head` in the previous position of the last element.

In other words, given the appropriate circumstances, we wish to transform the above invocation of `append` into something like

```
(begin
  (when (is (length below) > 0)
    (array-set! list (last below) 0)
    (array-set! list head (length below)))
  (quicksort! below)
  (quicksort! above)
  list)
```

Now the most difficult part is to specify what exactly is to be meant by *appropriate circumstances*. As in the case of `array-passing` convention, our transformation needs to decide that the result of the function can overwrite its argument (which in general depends on the way the argument is used by the caller).

Assuming that this is indeed the case, we shall transform all the functions that are used by `quicksort` – most notably, the `Hoare-partition` function.

Again, we need to determine when it is OK, for example, to replace references to `swap` with references to `swap!`, and this can actually be quite tricky.

For example, suppose that we had a mutating variant of `alter`

```
(define (alter! n-th #;in array #;with replacement)
  (array-set! array replacement n-th)
  array)
```

and wanted to use it to mechanically derive the definition of `swap!` from the definition of `swap`. The naive substitution would yield the following definition:

```
(define (swap!/incorrect i-th #;with j-th #;in array)
  (alter! i-th (alter! j-th array (array-ref array i-th))
         (array-ref array j-th)))
```

The problem is that, since the order of evaluation of arguments is unspecified, it is possible (and likely) that `(alter! j-th array (array-ref array i-th))` will be evaluated before `(array-ref array j-th)`, causing the latter to refer to a modified object. As a result, instead of swapping, we will only duplicate the `j-th` element. (Even worse, if the evaluation order is reversed, the problem may remain unnoticed!)

Of course, the straightforward solution to this problem is to extract all references to arrays before any mutation takes place:

```
(define (swap! i-th #;with j-th #;in array)
  (let ((array/i (array-ref array i-th))
        (array/j (array-ref array j-th)))
    (alter! i-th (alter! j-th array array/i) array/j)))
```

The derivation of the mutating counterpart of `Hoare-partition` seems extremely straightforward: we simply need to replace a reference to `list-ref` with a reference to `array-ref`, a reference to `swap` with reference to `swap!`, a reference to `length` with a reference to `array-length`, and a reference to `split-at` with a reference to `split-at!` that could be defined in the following way:

```
(define (split-at! array index)
  (values (slice array 0 index)
          (slice array index (- (array-length array)
                                index))))
```

The result of the transformation looks as follows:

```
(define (Hoare-Partition! condition array)

  (define (Parts! array back front)
    (cond ((is front >= (array-length array))
           (split-at! array back))

          ((condition (array-ref array front))
           (Parts! (swap! back front array) (+ back 1)
                                            (+ front 1)))
          (else
           (Parts! array back (+ front 1)))))

  (Parts! array 0 0))
```

Note that since the `array` argument in the `Parts!` helper function doesn't change between the calls, and could therefore be removed.

A much more puzzling question is: what makes this transformation so straightforward? Unfortunately, we have no answer for it. The fact is, that the code for `Hoare-partition` was itself derived from the code that was based on array slices.

The transformation of `quicksort` is somewhat more complex. It exploits certain properties regarding the memory layout of allocated objects that we talked about earlier. We hope that the intended meaning can be inferred from the names of predicates that are used to express these properties.

```
(define (Quicksort! array)
  (if (is (array-length array) = 0)
      array
  ;else
      (let ((head (array-ref array 0))
            (tail (slice array 1 (- (array-length array) 1))))
        (let* ((below above (Hoare-Partition! (is _ < head) tail)))
          (assert (and (continuous-region? below above)
                       (same-region? (rejoin below above) tail))
                       (same-region? array (Quicksort! array)))
            (when (is (array-length below) > 0)
              (array-set! array (array-last below) 0)
              (set! below (slice array 0 (array-length below)))
              (array-set! array head (array-length below)))
            (Quicksort! below)
            (Quicksort! above)
            array))))
```

where `array-last` is defined as

```
(define (array-last array)
  (array-ref array (- (array-length array) 1)))
```

The resemblance between `Quicksort!` and `quicksort` is not easy to see, and therefore the transformation is far from obvious.

The `()` pattern

The pattern `()` is mapped to the condition `(is (array-length array) = 0)`. Moreover, the result `'()` is mapped to the value of `array`. Unlike in the case of the functional variant, we cannot simply return any empty array (in particular, we cannot allocate a new empty array, or return some generic object that would represent an empty array).

Our optimization would need to figure out that the result we're returning is actually contained in one of its arguments. Deciding which argument it is supposed to be is not an easy task in general (on the other hand, in this particular case we don't have many candidates to consider).

The `(head . tail)` pattern

The clause `((head . tail) <template>)` is mapped to the

```
(let ((head (array-ref array 0))
      (tail (slice array 1 (- (array-length array) 1))))
  <template*>)
```

expression, and

```
(let* ((below above (Hoare-partition (is _ < head) tail)))
  <body>)
```

is simply mapped to

```
(let* ((below above (Hoare-Partition! (is _ < head) tail)))
  <body*>)
```

Of course, in the case of actual code, all the `let`, `let*`, `match` and `is` forms would be expanded to `if`, `lambda` and `call-with-value` forms prior to the transformation.

Turning concatenation into imperative operations

The transformation of `‘(,@(quicksort below) ,head ,@(quicksort above))` has already been discussed to some extent, although the actual code differs slightly from what we have anticipated – its particular form is the following sequence of operations:

```
(when (is (array-length below) > 0)
  (array-set! array (array-last below) 0)
  (set! below (slice array 0 (array-length below)))
  (array-set! array head (array-length below)))
(Quicksort! below)
(Quicksort! above)
array
```

This sequence of operations definitely begs for some explanation. In order to do that, let's now explicitly state the principle that implicitly drove the research throughout the previous chapter:

> if an output of a function is a collection containing no more elements than are contained in the collections of some of the function's arguments, and there are no references to the values of those arguments at any later execution point of that program (later than the invocation of that function, that is), then it should be advised that the collections are laid out in a continuous region of memory. When this is done, then the storage occupied by input arguments can be re-used to store the value of the output.

From there, it follows that

> if an output of a function is a permutation of some of its inputs, and there are no references to that input at any later execution point of that program, then the permutation can be performed in place on that input (instead of performing a copy).

This conclusion is insufficient to perform the considered transformation. We need to resort to another principle:

> if the result of a function is obtained by appending some items to a collection whose order of elements is irrelevant from the point of view of the rest of the computation (that is, at the time of appending the collection can be treated as a set rather than sequence), and there is a sufficient amount of unused storage in front of the collection being appended to, then the computation can proceed by copying elements from the back of the sequence before the front of the sequence, and moving the appended items into the freed place.

The formulation of this principle may not be satisfactory: it can often be the case that there is some free space at the back of the collection being appended to, and there is no need to move the elements from the back before the front of the collection.

Apparently, this isn't the case in our example: we know that the `above` array (or – more precisely – the array obtained from `(quicksort above)`) is already in the right place, and – at the moment of transformation – we're unable to tell whether there is any free space following that array.

There is yet another problem with the treatment of the `append` function: the `below` array that is being appended to doesn't appear directly, but it is passed into `quicksort`.

However, the resulting code needs to perform the concatenation *before* the `Quicksort!` is called, and not after it is called.

We could therefore formulate the following principle:

> if an array is a result of a function that produces a *deterministic permutation* of its argument, and this array is the first argument to `append`, then the optimized counterpart of the expression can perform concatenation prior to execution of the optimized version of the function.

Intuitively, the validity of this principle stems from the requirement that the function must be a deterministic permutation, which means that when it is passed any permutation of a collection of elements, then the order of elements in its result will always be the same.

Now this principle may seem very particular, as if it were cut out especially to tackle our problem, and we admit that this was indeed the case.

It would of course be much better if we had a more general principle from which the above one would follow.

To sum up, in order to transform `(append (quicksort below) (cons head (quicksort above))` like we want to, we must show that:

1. the value of the whole expression can replace the input argument, i.e. `list`, because `(equal? (length list) (length (quicksort list)))`;
2. the value of `above` and/or `(quicksort above)` already occupies the desired position in memory;
3. there is enough unused storage before `below` to perform the concatenation by swapping elements from the back to the front and inserting the elements that are not in their final position (i.e. `head`) at the back;
4. from the point of view of the rest of the computation, `before` can be treated as a set, i.e. the order of the elements contained in it is irrelevant;
5. `quicksort` is a deterministic permutation.

As we have shown in chapter 6, the requirement 1 can only be decided upon the usage of the function in the rest of the program (however, we could always make a copy prior to the call to a function, assuring that the in-place optimization can be done).

7.3 Conclusions and future work

We believe that we have identified the conditions that are needed to transform the piece of functional code into its imperative counterpart that avoids generating garbage.

We admit that these conditions are very particular, and that it would be more desirable to have a more general set of optimizations that would be able to handle the case of quicksort along with other cases.

Also, we've found that it is rather difficult to translate these rules into a working Scheme code, so unlike in previous chapters, we give no working Scheme code for carrying out the transformation. Doing so would probably require the development of some framework for expressing general optimization/transformation rules. We hope this to be done in the future.

While this work focused particularly on sorting, we hope it didn't loose its broader goal, which is to allow to take the burden of data structure selection off from programmers.

Certainly, at this stage there's still a lot to be done. We admit that the methodology developed in this work, i.e. writing both functional and imperative solutions and then figuring out the transformation from the former to the latter, isn't particularly effective. We believe however, that over

time some more swift methods for optimizing programs will be developed. Furthermore, this effort needs only be done by people who specialize in such optimization, allowing the majority of language users to express their programs in a way that is just convenient, without having to worry too much about their performance.

We also believe that the approach to program optimization presented in this work may prove itself much more scalable than the more conventional approach, where programmers achieve speed-ups by modifying their original programs, because the same optimization could be used by more than just a one program.

Of course, when it comes to optimization, it is reasonable to focus on the most common cases first. For this reason, we suggest that it might be a good idea to extend our transformation to handle matrix operations (where a matrix is to be represented as a list of lists of equal length).

Moreover, we believe that the development of a formal system for reasoning about the time and space complexity of functions would be an important step towards making an automatic system for optimizing programs.

Another idea to pursue is to have the compiler deduce the entropy of certain variables to minimize the amount of bits that are used to represent them. This way, it should sometimes be possible to have a couple of values stored in a single register.

An even more radical idea is to feed the compiler with both the program and the description of the instruction set of the target processor, and have it automatically come up with a sequence of instructions that is (in some sense) isomorphic with the original program.

Judging by the number of conferences, functional programming techniques have recently been getting more recognition in the industry. A likely reason for this state of affairs is that computer hardware is cheap and fast enough enough to run programs that were written with readability and maintainability in mind, rather than performance – and indeed, the costs of programmers' mistakes and overlong development time often exceeds the costs of hardware by a few orders of magnitude. It is therefore reasonable to search for techniques of increasing software reliability, even if the price to pay is increased consumption of computing resources.

However, the availability of cheap processing power should not be an excuse for wasting it. We believe (and hope that we managed to show this to some extent in this work) that functional programs may benefit from better maintainability without any performance loss whatsoever, and that the process of programming could be simplified further by moving the burden of dealing with data structures from the programmer to the compiler. As programming is becoming more and more popular an activity, but programmers are not necessarily becoming more competent, we suspect that this could even prevent some catastrophes in the future.

Part III

Appendices

Appendix A

Non-standard functions

For the purpose of reference, we provide the definitions of non-standard functions that were described in chapter 2. The `e.g.` form can be treated as a comment (although it could also be used to express unit tests in actual programs).

```
(define (fold-left op e . ls)
  (match ls
    (((heads . tails) ...)
     (apply fold-left op (apply op e heads) tails))
    (_
     e)))

(define (fold-right op e . ls)
  (match ls
    (((heads . tails) ...)
     (apply op `(,@heads ,(apply fold-right op e tails))))
    (_
     e)))

(define (every property? . ls)
  (apply fold-left (lambda (result . elements)
                     (and result (apply property? elements)))
                   #true
                   ls))

(e.g. (and (every even? '(2 4 6))
           (not (every even? '(1 2 3)))))

(define (any property? . ls)
  (apply fold-left (lambda (result . elements)
                     (or result (apply property? elements)))
                   #false
                   ls))
```

```
  (e.g. (and (any even? '(1 2 3))
             (not (any even? '(1 3 5)))))

  (define (member element set)
    (any (is _ equal? element) set))

  (e.g. (is 2 member '(1 2 3)))

  (define (subset? x y)
    (every (is _ member y) x))

  (e.g. (and (is '(1 2 3) subset? '(5 4 3 2 1))
             (isnt '(0 1 2) subset? '(2 3 4))))

  (define (same-sets? set . sets)
    (define (same-set? a b)
      (and (is a subset? b)
           (is b subset? a)))
    (every (is _ same-set? set) sets))

  (e.g. (and (same-sets? '(1 2 3) '(3 1 2))
             (not (same-sets? '(1 2 3) '(1 2 3 4)))))

  (define (union set . sets)
    (define (union a b)
      (fold-left (lambda (set element)
                   (if (is element member set)
                       set
                       `(,element . ,set)))
                 a
                 b))
    (fold-left union set sets))

(e.g. (same-sets? (union '(1 2 3) '(2 3 4)) '(1 2 3 4)))

  (define (intersection set . sets)
    (define (intersection a b)
      (fold-left (lambda (set element)
                   (if (is element member a)
                       `(,element . ,set)
                       set))
                 '()
                 b))
    (fold-left intersection set sets))

  (e.g. (same-sets? (intersection '(1 2 3) '(2 3 4)) '(2 3)))
```

The `for` loop used in chapter 3 is a macro that could be defined in the following way:

```
(define-syntax for
  (syntax-rules (in)
    ((for element in sequence . actions)
     (for-each (lambda (element) . actions)
               sequence))))
```

where `for-each` is defined as

```
(define (for-each action list)
  (unless (null? list)
    (action (car list))
    (for-each action (cdr list))))
```

The `range` function can be defined as

```
(define (range start end)
  (if (is start > end)
      '()
      `(,start . ,(range (+ start 1) end))))

(e.g.
 (range 1 10)
 ===> (1 2 3 4 5 6 7 8 9 10))

(define ((number/base base) list)
  (fold-left (lambda (number digit)
               (+ (* number base) digit))
             0
             list))

(e.g.
  ((number/base 2) '(1 0 0)) ===> 4
  ((number/base 10) '(1 0 0)) ===> 100)

(define ((digits/base base) number)
  (define (divide number digits)
    (if (= number 0)
        digits
    ;else
        (divide (quotient number base)
                `(,(modulo number base) . ,digits))))
  (divide number '()))
```

```
(e.g.
 ((digits/base 2) 4) ===> '(1 0 0)
 ((digits/base 10) 140) ===> '(1 4 0))

(define (extend-left list size fill)
  (assert (is (length list) <= size))
  (if (is (length list) < size)
      (extend-left `(,fill . ,list) size fill)
  ;else
      list))

(e.g. (extend-left '(1 2 3) 5 0) ===> (0 0 1 2 3))
```

The `machine-word-bytes` function used to implement the virtual machine from chapter 3 can be defined as follows (given some definite value of `MACHINE-WORD-SIZE`):

```
(define (machine-word-bytes value)
  (extend-left ((digits/base 256) value) MACHINE-WORD-SIZE 0))
```

Appendix B

Y-lining

As we noted earlier, since the definitions can in general refer to themselves, we ought to transform them to `lambda` expressions in the context of the `Z` combinator. However, we need to generalize it to allow an arbitrary number of mutually recursive functions to access one another. Following [41], we can transform the programs like:

```
(define even? (lambda (n)
                (or (= n 0)
                    (odd? (- n 1)))))

(define odd? (lambda (n)
               (and (not (= n 0))
                    (even? (- n 1)))))

(even? 5)
```

into expressions like

```
(let ((even? (lambda ((even? odd?)) even?))
      (odd? (lambda ((even? odd?)) odd?)))
  (let* ((combine (lambda (tuple)
                    `(,(lambda (n)
                         (or (= n 0)
                             ((odd? (tuple)) (- n 1))))
                      ,(lambda (n)
                         (and (not (= n 0))
                              ((even? (tuple)) (- n 1)))))))
         ((even? odd?) (Z combine)))
    (even? 5)))
```

Or, to put it more generally, we wish to transform

```
(define name-1 value-1)
(define name-2 value-2)
...
(define name-n value-n)
expression
```

into

```
(let ((name-1 (lambda ((name-1 name-2 ... name-n)) name-1))
      (name-2 (lambda ((name-1 name-2 ... name-n)) name-2))
       ...
      (name-n (lambda ((name-1 name-2 ... name-n)) name-n)))
  (let* ((combine (lambda (tuple)
                    `(,value-1* ,value-2* ... ,value-n*)))
         ((name-1 name-2 ... name-n) ((Z combine))))
    expression))
```

where `value-i*` is obtained from `value-i` by replacing all references to `name-k` with expressions `(name-k (tuple))`. Also, the symbol `tuple` must not occur free in `value-k`, nor should `name-k` consist of the symbol `tuple`.

Using the definition of `substitute` from chapter 5 (page 75), we can define the transformation in the following way:

```
(define (Y-line program)
  (match program
    ((('define names values) ... expression)
     (let* (((Y X T) (map unique-symbol '(Y X T)))
            (references (map (lambda (name)
                               `(,name (,T)))
                             names))
            (values* (map (lambda (value)
                            (substitute names references value))
                          values)))
       `(let* ((,Y (lambda (f)
                     (let ((R (lambda (g)
                                (lambda ()
                                  (f (g g))))))
                       (R R))))
               ,@(map (lambda (name)
                        `(,name (lambda (,names) ,name)))
                      names)
               (,X (lambda (,T) (list . ,values*)))
               (,names ((,Y ,X))))
          ,expression)))))
```

Needless to say, the programs obtained this way are terribly inefficient.

Appendix C

Macro expansion

Prior to evaluation, we need to convert all the special forms like `let` or `and` into a program consisting solely of primitive forms `lambda` and `if`.

The R^{5}RS specification of Scheme provides a special language for defining new syntactic extensions, called `syntax-rules`.

While there are free implementations available, we believe that it is too complex for our purpose. Instead we are going to propose a language that is similar but slightly simpler.

As in the case of `syntax-rules`, we shall be writing down our macros using patterns and templates. For example, we'd like to be able to define the core Scheme macros in the following way:

```
(define core-macros
  '((('let ((name value) ...)
       . body)
     (('lambda (name ...) . body) value ...))

    (('let* () . body)
     ('begin . body))

    (('let* ((name-1 value-1)
             (name-2 value-2) ...)
       . body)
     ('let ((name-1 value-1))
       ('let* ((name-2 value-2) ...)
         . body)))

    (('and)
     #true)

    (('and last)
     last)
```

```
(('and first . rest)
 ('if first ('and . rest) #false))

(('or)
 #false)

(('or last)
 last)

(('or first . rest)
 ('let ((result first))
   ('if result result ('or . rest))))))
```

C.1 Binding patterns

In order to be able to apply these macros, we need to be able to *bind* patterns to forms. We are going to represent a binding using an association list (if there is no binding, we expect the function to return `#false`). We expect that

```
(e.g. (same-sets? (bind '('let ((names values) ...) . body)
                        '(let ((a 5) (b 10)) (+ a b)))
                  '((names a b) (values 5 10) (body (+ a b)))))
```

and that

```
(e.g. (not (bind '(a b c) '(1 2))))
```

From the examples above one can infer the following:

1. literals are represented using `quote`d symbols,
2. variables are represented using regular symbols,
3. the `...` symbol has a special meaning: it is used to represent a repetition of zero or more forms that precede it. Consequently, it causes symbols to capture lists of values, rather than individual values.

If a symbol appears more than once in the pattern, we expect both occurrences to be `equal?`.

```
(e.g. (bind '(a b a) '(1 2 1)))
```

but

```
(e.g. (not (bind '(a b a) '(1 2 3))))
```

The code here is heavily inspired by the implementation from [63].

```
(define (bind pattern #;to form . bound-variables)
  (match pattern
    (('quote literal)
     (and (equal? form literal)
          bound-variables))

    ((repetition '... . remaining)
     (bind-sequence repetition remaining form
                    bound-variables))

    ((head/pattern . tail/pattern)
     (match form
       ((head/form . tail/form)
        (let ((bound (apply bind head/pattern head/form
                            bound-variables)))
          (and bound
               (apply bind tail/pattern tail/form bound))))
       (_
        #false)))

    (_
     (if (symbol? pattern)
       (merge-bindings `((,pattern . ,form) bound-variables))
     ;else
       (and (equal? pattern form)
            bound-variables)))))
```

where

```
(define (merge-bindings bindings . bindings*)
  (define (merge-bindings a b)
    (and a b
         (fold-left (lambda (bindings (key . value))
                      (and bindings
                           (cond ((assoc key bindings)
                                  => (lambda ((key . value*))
                                       (and (equal? value value*)
                                            bindings)))
                                 (else
                                  `((,key . ,value)
                                    . ,bindings)))))
                    a
                    b)))
  (fold-left merge-bindings bindings bindings*))
```

We have used the feature of the `cond` variable that we didn't describe in chapter 2: if the condition is followed by the `=>` symbol, then the following expression must be a function of one argument.

If the value of the condition is other than `#false`, then it is passed to that function, yielding the value of the `cond` expression.

As we can see, the definition of `bind` is rather straightforward: we must only consider seven cases. The first one is the occurrence of a literal, which is compared using `equal?`.

The second is a pattern followed by an ellipsis. Since it is a bit complex, it is handled by a separate function called `bind-sequence` that is explained below [1].

The third is when pattern is a pair. In this case, the form being pattern-matched must also be a pair, and we should be able to bind the head of the pattern with the head of the form, and the tail of the pattern with the tail of the form, unifying the bindings.

Otherwise, the pattern is either a literal (such as a number) or a symbol. If it is a literal, it is compared with the form using the `equal?` predicate. Otherwise it may either either be bound or unbound. If it is bound, then form must be `equal?` to the bound value. Otherwise, a new binding is added to the `bound-variables`.

Adding support for ellipses is a bit tricky. When we encounter the `...` symbol, we need to make sure that we're both able to match some prefix of the form so that each element matches the pattern preceding the `...` symbol, and that the part of the form that didn't get into the prefix matches the remainder of the pattern.

The `bind-sequence` function will need to call `bind` recursively on some prefix of the form being pattern-matched, and then zip the resulting bindings (note that the `zip-bindings` function requires that the order of bindings is the same for each invocation of `bind`)

```
(define (zip-bindings list-of-bindings)
  (match list-of-bindings
    ((((names . values) ...) ...)
     (assert (apply eq? names))
     (match names
       ((names . _)
        (apply map list names values))
       (()
        '()))))))
```

[1] Although we wrote the pattern as `(repeated '... . remaining)`, the `match` macro that we used to write this code doesn't treat the quoted ellipses properly. Instead, in the actual code we had to resort to the feature called *guarded patterns*: instead of `'...`, we wrote `(? ...?)`, where `(define (...? x) (eq? x '...))`.

```
(e.g.
 (zip-bindings '(((a . 1) (b . 2) (c . 3))
                 ((a . 4) (b . 5) (c . 6))
                 ((a . 7) (b . 8) (c . 9))))
 ===> ((a 1 4 7) (b 2 5 8) (c 3 6 9)))
```

Since – as we mentioned earlier – the presence of ellipses may cause ambiguous match, we are going to need to test various possible matches, until we find the satisfying one.

```
(define (bind-sequence repeated-pattern remaining-pattern
                       form bound-variables)
  (define (successful-match? prefix suffix)
    (let* ((bindings (map (lambda (form)
                            (bind repeated-pattern form))
                          prefix))
           (zipped (zip-bindings bindings))
           (merged (merge-bindings bound-variables zipped)))
      (and merged (apply bind remaining-pattern suffix
                         merged))))

  (let* ((limit (prefix-length (lambda (constituent)
                                 (bind repeated-pattern
                                       constituent))
                               form))
         (prefix rest (split-at form limit)))
    (carry #;from prefix #;to rest
           #;until successful-match?)))
```

The `carry` function is used for testing smaller and smaller prefixes (and – accordingly – longer and longer suffixes) until it finds a division that satisfies the condition:

```
(define (carry #;from prefix #;to suffix #;until success?)
  (let ((result (success? prefix suffix)))
    (if (or result (null? prefix))
        result
    ;else
        (let (((initial ... last) prefix))
          (carry #;from initial #;to `(,last . ,suffix)
                 #;until success?)))))
```

The `prefix-length` function returns the number of initial elements on the list that satisfy a given condition:

```
(define (prefix-length condition? l)
  (define (traverse l n)
    (match l
      ((head . tail)
       (if (condition? head)
           (traverse tail (+ n 1))
       ;else
           n))
      (_
       n)))
  (traverse l 0))

(e.g.
  (prefix-length even? '(2 4 6 7 8 9)) ===> 3)
```

C.2 Filling templates

Once the pattern is matched and the appropriate names are associated with corresponding values, the association can be used for filling templates.

We wish that the patterns and templates are symmetrical: if a quoted symbol appears in the template, it should be transformed into literal symbol. Otherwise if a bound symbol appears on the association list, then it should be replaced with the associated value.

An interesting case is when a symbol is not quoted, and it does not appear on the association list: drawing inspiration from `syntax-rules`, we replace it with a new symbol that is guaranteed not to clash with any other symbol used in the program.

For this reason, we need to be able to identify all the symbols that are used in the template:

```
(define (used-symbols expression)
  (match expression
    (('quote literal)
     '())

    ((repeated '... . rest)
     (union (used-symbols repeated)
            (used-symbols rest)))

    ((head . tail)
     (union (used-symbols head)
            (used-symbols tail)))
```

```
      (_
       (if (symbol? expression)
           `(,expression)
       ;else
           '())))))

  (e.g. (same-sets? (used-symbols '(a ... a b ... c 'c 'd))
                    '(a b c)))
```

We can extend the bindings to be filled in a template with mappings from the used symbols that are not bound to freshly generated symbols (that are guaranteed to be distinct from every other symbol present in the program):

```
(define (fill template #;with bindings)
  (let* ((missing (difference (used-symbols template)
                              (map (lambda ((key . value))
                                     key)
                                   bindings)))
         (bindings `(,@(map (lambda (symbol)
                              `(,symbol . ,(unique-symbol symbol)))
                            missing) ,@bindings)))
    (fill-template template bindings)))
```

The `used-symbols` can be used to generate missing bindings while filling the template:

```
  (define (fill-template template #;with bindings)
    (match template
      (('quote literal)
       literal)

    ((repeated '... . rest)
     `(,@(fill-sequence repeated bindings)
       . ,(fill-template rest #;with bindings)))

    ((head . tail)
     `(,(fill-template head #;with bindings)
       . ,(fill-template tail #;with bindings)))

    (_
     (cond ((and (symbol? template)
                 (assoc template bindings))
            => (lambda ((key . value))
                 value))
```

```
          (else
           template)))))
```

As before, the most difficult part is the treatment of ellipses. Given a template that is followed by an ellipsis, we need to identify all the symbols that appear in that template (but we do not count symbols which are quoted):

Subsequently, we need to be able to unzip symbols that appear in patterns directly preceding the ellipsis. To be more precise, we need to convert bindings into a sequence of bindings:

```
(define (unzip-bindings bindings keys)
  (let* ((unzipped (filter (lambda ((key . value))
                             (is key member keys))
                           bindings))
         (((names . values) ...) unzipped))
    (map (lambda (singular-values)
           `(,@(map (lambda (name value)
                      `(,name . ,value))
                    names singular-values)
             ,@bindings))
         (transpose values))))

  (e.g.
   (unzip-bindings '((a 1 2 3) (b 1 2 3) (c 1 2 3) (d . 4)) '(a c e))
    ===> (((a . 1) (c . 1) (a 1 2 3) (b 1 2 3) (c 1 2 3) (d . 4))
          ((a . 2) (c . 2) (a 1 2 3) (b 1 2 3) (c 1 2 3) (d . 4))
          ((a . 3) (c . 3) (a 1 2 3) (b 1 2 3) (c 1 2 3) (d . 4))))
```

where `transpose` can be defined as

```
(define (transpose list-of-lists)
  (if (null? list-of-lists)
      '()
  ;else
      (apply map list list-of-lists)))

(e.g. (transpose '((1 2 3)
                   (4 5 6))) ===> ((1 4)
                                   (2 5)
                                   (3 6)))
```

Given these two operations, filling a sequence is rather straightforward:

```
  (define (fill-sequence template bindings)
    (let* ((symbols (used-symbols template))
           (binding-sequences (unzip-bindings bindings symbols)))
      (map (lambda (bindings)
             (fill-template template bindings))
           binding-sequences)))
```

C.3 Expansion

Having `bind` and `fill`, we can now construct our expander:

```
(define (expand expression macros)

  (define (transform expression)
    (let ((result (any (lambda ((pattern template))
                         (let ((bindings (bind pattern
                                               expression)))
                           (and bindings
                                `(,bindings ,template))))
                       macros)))
      (match result
        ((bindings template)
         (fill template bindings))

        (_
         expression))))

  (define (expand expression)
    (match expression
      (('quote _)
       expression)

      (('lambda args body)
       `(lambda ,args ,(expand body)))

      (('if condition then else)
       `(if ,(expand condition)
            ,(expand then)
            ,(expand else)))

      ((operator . operands)
       (let ((transformed (fix transform expression)))
         (if (equal? expression transformed)
             `(,(expand operator) . ,(map expand operands))
         ;else
             (expand transformed))))
```

```
      (_
       expression)))

  (expand expression))
```

where `fix` is defined to iterate until reaching a fixed point:

```
(define (fix function argument)
  (let ((value (function argument)))
    (if (equal? value argument)
        value
    ;else
        (fix function value))))
```

Given the definition of `core-macros` provided at the beginning of this appendix, we can see that

```
(e.g.
  (expand '(let* ((a 5) (b (* a 2)))
             (or (> a b)
                 (+ a b)))
          core-macros) ===> ((lambda (a)
                               ((lambda (b)
                                  (begin
                                    ((lambda (##result#1)
                                       (if ##result#1
                                           ##result#1
                                           (+ a b)))
                                     (> a b))))
                                (* a 2)))
                             5))
```

One can see ha there's a symmetry between the patterns and the templates in the definition of `core-macros`. This could prompt someone to equip the `expand` function with the facility of reverting the expansion.

We have indeed made a successful attempt in this direction, although it wasn't mature enough to incorporate it here.

Appendix D

Hudak quicksort

Below we provide an implementation of Hudak's functional variant of quicksort that was originally given in [45], translated directly to Scheme. We failed to comprehend the idea of that implementation in the extent that would allow to fix it. As one can see, we provide our own indexing function called `ref`, expressed using the built-in `list-ref` function.

```
(define (ref v n)
  (list-ref v (- n 1)))
```

We also provide our own implementation of the `update` function, whose semantics is explained in the paper.

```
(define (update array index element)
  (alter (- index 1) array element))
```

where `alter` is defined as in chapter 7 (page 117).

The rest of the program is a straightforward translation of Hudak's code.

```
(define (quicksort v)
  (qsort v 1 (length v)))

(define (qsort v left right)
  (if (is left >= right)
      v
      (scan-right v            (+ left 1) right
                  (ref v left)       left right)))
```

```
(define (scan-right v l r pivot left right)
  (cond ((= l r)
         (finish (update v l pivot)
                 l left right))
        ((is (ref v r) >= pivot)
         (scan-right v l (- r 1) pivot left right))
        (else
         (scan-left (update v l (ref v r))
                    (+ l 1) r pivot left right))))

(define (scan-left v l r pivot left right)
  (cond ((= l r)
         (finish (update v l pivot)
                 l left right))
        ((is (ref v l) <= pivot)
         (scan-left v (+ l 1) r pivot left right))
        (else
         (scan-right (update v r (ref v l))
                     l (- r 1) pivot left right))))

(define (finish v mid left right)
  (qsort (qsort v left (- mid 1)) (+ mid 1) right))
```

One can easily see that the code fails to work as expected:

```
(e.g.
 (quicksort '(4 3 9 8 7 1 2 6 5))
===> (4 4 4 4 7 7 7 7 9))
```

Appendix E

The compiler

Below is the full source code of the compiler from chapter 4. It has been tested with Guile 2.0.11[1].

```
(use-modules (grand scheme) (grand symbol) (srfi srfi-88))


(define primitive-operators
  '((+ pass+)
    (- pass-)
    (* pass*)
    (/ pass/)
    (% pass%)
    (&& pass&&)
    (|| pass||)
    (^ pass^)
    (<< pass<<)
    (>> pass>>)
    (= pass=)
    (< pass<)
    (<= pass<=)
    (<> pass<>)
    (>= pass>=)
    (> pass>)))

(define (comparison? operator)
  (any (lambda ((left right))
         (or (equal? operator left)
             (equal? operator right)))
       mutually-negating-comparisons))
```

[1] https://www.gnu.org/software/guile/

```
(define (compound? expression)
  (pair? expression))


(define mutually-negating-comparisons
  '((< >=)
    (<= >)
    (= <>)))


(define (primitive-operator? operator)
  (any (lambda ((primop passing-function))
         (equal? operator primop))
       primitive-operators))


(define (inversion comparison)
  (any (lambda ((left right))
         (or (and (equal? left comparison)
                  right)
             (and (equal? right comparison)
                  left)))
       mutually-negating-comparisons))


(define (passing-function function-name)
  (or (any (lambda ((operator passing-name))
             (and (equal? operator function-name)
                  passing-name))
           primitive-operators)
      (and (comparison? function-name)
           (symbol-append 'pass function-name))
      (symbol-append 'pass- function-name)))


(define (expression-name expression)
  (match expression
    ((head . tail)
     (if (or (primitive-operator? head)
             (comparison? head))
         (apply symbol-append
                (intersperse head (map expression-name tail)))
         (apply symbol-append (expression-name head) '/
                (intersperse ': (map expression-name tail)))))
    (_
     (pass expression ->string string->symbol))))
```

```
(define original-name
  (let ((number 0))
    (lambda base
      (match base
        (()
         (set! number 0))
        ((base)
         (set! number (+ number 1))
         (string->symbol (string-append
                          (->string (expression-name base))
                          "/"
                          (->string number)))))))))

(define (passing-arguments arguments names final)
  (assert (= (length arguments) (length names)))
  (match arguments
    (()
     (assert (null? names))
     final)
    ((argument . next)
     (let (((name . names) names))
       (if (compound? argument)
           (passing-arguments next names
                              (passing argument
                                       `(lambda (,name) ,final)))
       ;else
           (passing-arguments next names final))))))

(define (passing expression continuation)
  (match expression
    (('quote _)
     `(,continuation ,expression))

    (('if <condition> <then> <else>)
     (let ((result (original-name <condition>)))
       (passing <condition> `(lambda (,result)
                               (if ,result
                                   ,(passing <then> continuation)
                                   ,(passing <else> continuation))))))

    (('lambda <args> <body>)
     `(,continuation (lambda (,@<args> return)
                       ,(passing <body> 'return))))
```

```
    ((function . arguments)
     (let ((simple-arguments (map (lambda (argument)
                                    (if (compound? argument)
                                        (original-name argument)
                                        argument))
                                  arguments)))
       (passing-arguments arguments simple-arguments
                          `(,(passing-function function)
                            ,@simple-arguments
                            ,continuation))))

    (_
     `(,continuation ,expression))))


(define (passing-program program)
  (let ((('define names functions) ... expression) program))
    `(,@(map (lambda (name function)
               (let ((('return pass-function) (passing function 'return)))
                 `(define ,(passing-function name) ,pass-function)))
             names functions)
      ,(passing expression 'exit))))


(define (pass= a b return)
  (return (= a b)))


(define (pass- a b return)
  (return (- a b)))


(define (pass* a b return)
  (return (* a b)))


(e.g.
 (begin
   (original-name)
   (passing-program '((define !
                        (lambda (n)
                          (if (= n 0)
                              1
                              (* n (! (- n 1))))))
                      (! 5))))
 ===>
```

```
 ((define pass-!
    (lambda (n return)
      (pass= n
             0
             (lambda (n=0/1)
               (if n=0/1
                   (return 1)
                   (pass- n
                          1
                          (lambda (n-1/3)
                            (pass-!
                             n-1/3
                             (lambda (!/n-1/2)
                               (pass* n !/n-1/2 return)))))))))
  (pass-! 5 exit)))

(define (passing-function-label name)
  (let (((content) (symbol-match "^pass-(.+)$" name)))
    (string->keyword content)))

(define (label . parts)
  (string->keyword (apply string-append (map ->string parts))))

(define new-label
  (let ((label-counter 0))
    (lambda parts
      (cond ((null? parts)
             (set! label-counter 0))
            (else
             (set! label-counter (+ label-counter 1))
             (apply label `(,@parts - ,label-counter)))))))

(define (maybe-register source)
  (cond ((symbol? source)
         `(,source))
        ((pair? source)
         (first source))
        (else
         '())))

(define (instruction-registers/read instruction)
  (match instruction

    (([target] '<- source)
     (union `(,target) (maybe-register source)))
```

```
    ((target '<- register/value/location)
     (maybe-register register/value/location))

    ((target '<- left x right)
     (union (maybe-register left) (maybe-register right)))

    ((target '<- operator operand)
     (maybe-register operand))

    (('if left >?< right 'goto target)
     (union (maybe-register left) (maybe-register right)
            (maybe-register target)))

    (('push register/value)
     (maybe-register register/value))

    (('goto register/value)
     (maybe-register register/value))

    (_
     '())))

(define (instruction-registers/modified instruction)
  (match instruction

    (('pop register)
     `(,register))

    ((target '<- . _)
     `(,target))

    (_
     '())))

(define ((operation-from? actions) pass-@)
  (any (lambda ((@ pass-?))
         (equal? pass-@ pass-?))
       actions))

(define ((operator-from actions) pass-@)
  (let ((((@ pass-@) (find (lambda ((% pass-%))
                             (equal? pass-@ pass-%))
                           actions)))
    @))

(define (defined-function? operator)
  (symbol? operator))
```

```
(define sign (operator-from primitive-operators))

(define (used-registers machine-code)
  (match machine-code

    (()
     '())

    ((instruction . rest)
     (let ((read-registers (instruction-registers/read instruction))
           (modified-registers (instruction-registers/modified instruction)))
       (difference (union read-registers (used-registers rest))
                   (difference modified-registers read-registers))))))

(define (passing-program->assembly program/CPS)
  (let ((('define names passing-functions)
         ...
         expression) program/CPS))

    (define (argument-names function-name)
      (any (lambda (name ('lambda (arguments ... return) . _))
             (and (equal? name function-name)
                  arguments))
           names passing-functions))

    (define (assembly expression/cps registers)

      (match expression/cps
        (('lambda args body)
         (let ((label (new-label 'lambda)))
           `((result <- ,label)
             ,label
             ,@(assembly body registers))))

        (('return value)
         `((result <- ,value)
           (goto return)))

        ((pass<?> a b ('lambda (a<?>b) ('if a<?>b
                                        <then>
                                        <else>)))
          (let ((else (new-label 'else))
                (registers (union registers
                                  (maybe-register a)
                                  (maybe-register b))))
            `((if ,a ,(inversion (sign pass<?>)) ,b goto ,else)
              ,@(assembly <then> registers)
              ,else
              ,@(assembly <else> registers))))
```

```
    ((operator . operands)
     (call operator operands registers))))

(define (call operator operands registers)
  (cond ((primitive-operation? operator)
         (call-primitive operator operands registers))

        ((defined-function? operator)
         (call-defined operator operands registers))

        #;((anonymous-function? operator)
         (call-anonymous operator operands registers))))

(define (call-primitive operator operands registers)
  (let (((left right continuation) operands))
    (match continuation

      (('lambda (result) body)
       `((,result <- ,left ,(sign operator) ,right)
         ,@(assembly body (union registers
                                 (maybe-register left)
                                 (maybe-register right)
                                 `(,result)))))

      (_ ;; a ``return'' continuation
       `((result <- ,left ,(sign operator) ,right)
         (goto ,continuation))))))

(define (call-defined function arguments registers)

  (define (save registers)
    (map (lambda (register)
           `(push ,register))
         registers))

  (define (restore registers)
    (map (lambda (register)
           `(pop ,register))
         (reverse registers)))

  (define (pass values function)
    (let ((names (argument-names function)))
      (map (lambda (name value)
             `(,name <- ,value))
           names values)))
```

```
  ;; body of ‘call-defined’ begins here
  (let (((arguments ... continuation) arguments)
        (entry (passing-function-label function)))
    (match continuation

      ((’lambda (result) body)
       (let* ((proceed (new-label ’proceed))
              (sequel (assembly body registers))
              (registers (intersection registers
                                       (used-registers sequel))))
         ‘(,@(save registers)
           ,@(pass arguments function)
           (push return)
           (return <- ,proceed)
           (goto ,entry)
           ,proceed
           (pop return)
           ,@(restore registers)
           ,@sequel)))

      (_ ;; tail call optimization
       ‘(,@(pass arguments function)
         (goto ,(passing-function-label function)))))))

‘((return <- end:)
  ,@(assembly expression ’())
  ,@(append-map (lambda (name (’lambda args body))
                  ‘(,(passing-function-label name)
                    ,@(assembly body ’())))
                names passing-functions)
  end:
  (halt))))
```

```
(e.g.
 (begin
   (new-label)
   (passing-program->assembly
    '((define pass-!
        (lambda (n return)
          (pass= n 0
            (lambda (n=0/1)
              (if n=0/1
                  (return 1)
              ;else
                  (pass- n 1
                    (lambda (n-1/3)
                      (pass-! n-1/3
                        (lambda (!/n-1/2)
                          (pass* n !/n-1/2
                            return)))))))))
      (pass-! 5 return))))

 ===> ((return <- end:)
       (n <- 5)
       (goto !:)
     !:
       (if n <> 0 goto else-1:)
       (result <- 1)
       (goto return)
     else-1:
       (n-1/3 <- n - 1)
       (push n)
       (n <- n-1/3)
       (push return)
       (return <- proceed-2:)
       (goto !:)
     proceed-2:
       (pop return)
       (pop n)
       (result <- n * !/n-1/2)
       (goto return)
     end:
       (halt)))

(define (compile scheme-program)
  (assemble (passing-program->assembly
             (passing-program scheme-program))))
```

Appendix F

Overriding the core Scheme bindings

Readers who are familiar with the Scheme programming language probably noticed that the way it has been used in this work deviates from the standards defined in [66] and [70] because of the destructuring that can be performed in `lambda`, `let` and `let*` forms, as well as the possibility of creating *curried definitions* using the `define` form.

Since `define` and `lambda` forms are actually the core bindings, they cannot be in principle redefined. However, module systems present in some Scheme implementations allow to shadow the core bindings with some user-defined ones.

This section shows how this can be done with the module system available in Guile. The pattern matching is performed using the `(ice-9 match)` module that is shipped with Guile. It is a subset of the `(grand scheme)` glossary which is maintained by the author of this work[1].

```
(define-module (grand syntax)
  #:use-module (ice-9 match)
  #:use-module (srfi srfi-1)
  #:re-export (match)
  #:export (primitive-lambda)
  #:replace ((cdefine . define)
             (mlambda . lambda)
             (named-match-let-values . let)
             (match-let*-values . let*)))

(define-syntax mlambda
  (lambda (stx)
    (syntax-case stx ()
```

[1] `https://github.com/plande/grand-scheme`

```
      ((_ (first-arg ... last-arg . rest-args) . body)
       (and (every identifier? #'(first-arg ... last-arg))
            (or (identifier? #'rest-args) (null? #'rest-args)))
       #'(lambda (first-arg ... last-arg . rest-args) . body))


      ((_ arg body ...)
       (or (identifier? #'arg) (null? #'arg))
       #'(lambda arg body ...))


      ((_ args body ...)
       #'(match-lambda* (args body ...)
         (_ (error 'mlambda (current-source-location)
                   '(args body ...)))))
      )))


(define-syntax primitive-lambda
  (syntax-rules ()
    ((_ . whatever)
     (lambda . whatever))))


(define-syntax cdefine
  (syntax-rules ()
    ((_ ((head . tail) . args) body ...)
     (cdefine (head . tail)
       (mlambda args body ...)))
    ((_ (name . args) body ...)
     (define name (mlambda args body ...)))
    ((_ . rest)
     (define . rest))
    ))


(define-syntax match-let/error
  (syntax-rules ()
    ((_ ((structure expression) ...)
        body + ...)
     ((match-lambda* ((structure ...) body + ...)
        (_ (error 'match-let/error (current-source-location)
                  '((structure expression) ...)
                  expression ...)))
      expression ...))))
```

```
(define-syntax named-match-let-values
  (lambda (stx)
    (syntax-case stx ()
      ((_ ((identifier expression) ...) ;; optimization: plain "let" form
          body + ...)
       (every identifier? #'(identifier ...))
       #'(let ((identifier expression) ...)
           body + ...))

      ((_ name ((identifier expression) ...) ;; optimization: regular named-let
          body + ...)
       (and (identifier? #'name) (every identifier? #'(identifier ...)))
       #'(let name ((identifier expression) ...)
          body + ...))

      ((_ name ((structure expression) ...)
          body + ...)
       (identifier? #'name)
       #'(letrec ((name (mlambda (structure ...) body + ...)))
           (name expression ...)))

      ((_ ((structure expression) ...)
          body + ...)
       #'(match-let/error ((structure expression) ...)
                          body + ...))

      ((_ ((identifier identifiers ... expression)) body + ...)
       (every identifier? #'(identifier identifiers ...))
       #'(call-with-values (lambda () expression)
           (lambda (identifier identifiers ... . _)
             body + ...)))

      ((_ ((structure structures ... expression)) body + ...)
       #'(call-with-values (lambda () expression)
           (match-lambda*
               ((structure structures ... . _) body + ...)
             (_ (error 'named-match-let-values
                       (current-source-location)
                       'name)))))

      ((_ name ((identifier identifiers ... expression) body + ...))
       (and (identifier? #'name)
            (every identifier? #'(identifier identifiers ...)))
       #'(let ((name (lambda (identifier identifiers ...) body + ...)))
           (call-with-values (lambda () expression) name)))
```

```
      ((_ name ((structure structures ... expression) body + ...))
       (identifier? #'name)
       #'(let ((name (match-lambda* ((structure structures ...) body + ...)
                      (_ (error 'named-match-let-values
                                (current-source-location)
                                'name)))))
           (call-with-values (lambda () expression) name))))))

(define-syntax match-let*-values
  (lambda (stx)
    (syntax-case stx ()

      ((_ ((identifier expression) ...) ;; optimization: regular let*
          body + ...)
       (every identifier? #'(identifier ...))
       #'(let* ((identifier expression) ...)
           body + ...))

      ((_ ((identifier expression) remaining-bindings ...)
          body + ...)
       (identifier? #'identifier)
       #'(let ((identifier expression))
           (match-let*-values (remaining-bindings ...) body + ...)))

      ((_ ((structure expression) remaining-bindings ...)
          body + ...)
       #'(match-let/error ((structure expression))
                          (match-let*-values (remaining-bindings ...)
                                             body + ...)))

      ((_ ((identifier identifiers ... expression) remaining-bindings ...)
          body + ...)
       (every identifier? #'(identifier identifiers ...))
       #'(call-with-values (lambda () expression)
           (lambda (identifier identifiers ... . _)
             (match-let*-values (remaining-bindings ...)
                                body + ...))))

      ((_ ((structure structures ... expression) remaining-bindings ...)
          body + ...)
       #'(call-with-values (lambda () expression)
           (match-lambda* ((structure structures ... . _)
                           (match-let*-values (remaining-bindings ...)
                                              body + ...))
             (_ (error 'match-let*-values (current-source-location))))))
      )))
```

Bibliography

[1] Abelson, Harold and Gerald Jay Sussman with Julie Sussman, *Structure and Interpretation of Computer Programs*, Second Edition, MIT Press, 1996, ISBN 0-262-01153-0
`https://mitpress.mit.edu/sicp/full-text/book/book.html`

[2] Appel, Andrew W., *Compiling with Continuations*, Cambridge University Press, 1992

[3] Backus, John *Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs*, ACM Turing Award Lecture, 1977,
`http://worrydream.com/refs/Backus-CanProgrammingBeLiberated.pdf`

[4] Bagwell, Phil, *Fast Functional Lists, Hash-Lists, Deques and Variable Length Arrays*, 2002
`https://infoscience.epfl.ch/record/64410/files/techlists.pdf`

[5] Baker, Henry G., *Unify and Conquer (Garbage, Updating, Aliasing, ...) in Functional Languages*,
`http://www.pipeline.com/~hbaker1/Share-Unify.html`

[6] Baker, Henry G., *Shallow Binding Makes Functional Arrays Fast*,
`http://www.pipeline.com/~hbaker1/ShallowArrays.html`

[7] Ben-Ari, Mordechai, *Mathematical Logic for Computer Science*, Third Edition, Springer, 2012, ISBN 978-1-4471-4128-0

[8] Boyer, Robert S. and J Strother Moore, *A Computational Logic*, Academic Press, New York, 1979, ISBN 0-12-122950-5

[9] Boyer, Robert S. and J Strother Moore, *A Computational Logic Handbook*, Academic Press, New York, 1988

[10] Boyer, Robert S. and J Strother Moore, *Proving Theorems About LISP Functions*, Journal of the Association for Computing Machinery, Vol.

72, No. 1, January 1975, pp. 129-144
https://www.cs.utexas.edu/users/moore/publications/bm75.pdf

[11] Burstall, Rod, *Proving Properties of Programs by Structural Induction*, The Computer Journal, Vol. 12, No. 1, 1969, pp. 41-48
http://www.cse.chalmers.se/edu/year/2010/course/DAT140_Types/Burstall.pdf

[12] Burstall, Rod and John Darlington, *A Transformation System for Developing Recursive Programs*, Journal of the ACM, Vol. 21, No. 1, 1977, pp. 44-67
http://www.diku.dk/OLD/undervisning/2003e/235/Burstall-1977-TransSystem.pdf

[13] Chase, David R., *Garbage Collection and Other Optimizations*, Ph.D. Thesis, Rice University, August, 1987
https://scholarship.rice.edu/bitstream/handle/1911/16127/8900220.PDF?sequence=1&isAllowed=y

[14] Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, *Introduction to Algorithms*, Third Edition, MIT Press, 2009, ISBN 9780262533058

[15] Danvy, Olivier and Lasse R. Nielsen, *A First-Order One-Pass CPS Transformation*, BRICS Report Series RS-01-49, ISSN 0909-0878, Aarhus, December 2001
http://www.brics.dk/RS/01/49/BRICS-RS-01-49.pdf

[16] Dennett, Daniel C., *Intuition Pumps and Other Tools for Thinking*, New York, W. W. Norton & Company, 2013. ISBN 0393082067

[17] Dijkstra, Edsger, *On the Foolishness of "Natural Language Programming"*, Springer-Verlag London, 1997, ISBN 3-540-09251-X

[18] Dybvig, R. Kent, *Three Implementation Models of Scheme*, A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science, Chapel Hill, 1987
http://www.cs.indiana.edu/~dyb/pubs/3imp.pdf

[19] Dybvig, R. Kent, *The development of Chez Scheme*, in *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, 1-12, September, 2006
http://www.cs.indiana.edu/~dyb/pubs/hocs.pdf

[20] Dybvig, R. Kent, *The Scheme Programming Language*, 4th edition, MIT Press 2009, `http://www.scheme.com/tspl4/`

[21] Egner, Sebastian, *Scheme Request for Implementation 71: Extended LET-syntax for multiple values*, 2005
`https://srfi.schemers.org/srfi-71/`

[22] Fauconnier, Gilles and Mark Turner, *The Way We Think* Basic Books, 2002, ISBN 0-465-08786-8

[23] Feathers, Michael, *Working Effectively with Legacy Code*, Pearson Education, 2005, ISBN 0-13-117705-2

[24] Feeley, Marc, *90 minute Scheme to C compiler*, live tutorial recorded on video
`https://www.youtube.com/watch?v=Bp89aBm9tGU`
slides `http://churchturing.org/y/90-min-scc.pdf`

[25] Feeley, Marc, *Scheme Request for Implementation 39: Parameter objects*, 2003
`https://srfi.schemers.org/srfi-39/srfi-39.html`

[26] Feeley, Marc, *Scheme Request for Implementation 88: Keyword objects*, 2007
`https://srfi.schemers.org/srfi-88/`

[27] Felleisen, Matthias and Matthew Flatt, *Programming Languages and Lambda Calculi*, Utah CS7520 Version,
`https://www.cs.utah.edu/~mflatt/past-courses/cs7520/public_html/s06/notes.pdf`

[28] Felleisen, Matthias, Robert Bruce Findler, Matthew Flatt and Shriram Krishnamurthi, *How to Design Programs*, Second Edition, MIT Press, 2014
`http://www.ccs.neu.edu/home/matthias/HtDP2e/`

[29] Fowler, Martin et al. *Refactoring. Improving the Design of Existing Code* Pearson Education, 2000,

[30] Frege, Gottlob, *On Sense and Reference*, first published in *Zeitschrift für Philosophie unf philosophische Kritik*, 1892
in A. W. Moore (ed.) *Meaning and Reference*, Oxford University Press, 1993

[31] Friedman, Daniel P. and George Springer, *Scheme and the Art of Programming*, MIT Press, 1993, ISBN 0-262-69136-1

[32] Friedman, Daniel P., and Carl Eastlund, *The Little Prover*, MIT Press, 2015, ISBN 0-262-33056-3

[33] Friedman, Daniel P. and Matthias Felleisen, *The Little Schemer*, Fourth Edition, MIT Press 1996, ISBN 0-262-56099-2

[34] Friedman, Daniel P., Mitchell Wand and Christopher T. Haynes, *Essentials of Programming Languages*, Third Edition, MIT Press, 2008, ISBN 0-262-06279-8

[35] Futamura, Yoshihiko, Zenjiro Konishi and Robert Glück, *WSDFU: Program Transformation System Based on Generalized Partial Computation*, New Generation Computing, 20 (2002) 75-99
`http://repository.readscheme.org/ftp/papers/topps/D-489.pdf`

[36] Ghuloum, Abdulaziz, *An Incremental Approach to Compiler Construction*, Proceedings of the 2006 Scheme and Functional Programming Workshop, University of Chicago Technical Report TR-2006-06,
`https://github.com/panicz/inc/tree/master/docs`

[37] Godek, Panicz Maciej, *Scheme Request for Implementation: Syntactic combiners for binary predicates* (draft), 2017
`https://srfi.schemers.org/srfi-156/`

[38] Godek, Panicz Maciej, *Maszyna RAM i predykat Kleenego* (in Polish), notes from the course in Theory of Computation by Marcin Mostowski at the University of Warsaw (faculty of Philosophy), 2012
`https://github.com/panicz/writings/raw/master/archive/predykat-kleenego.pdf`

[39] Godek, Panicz Maciej, *A Pamphlet against R*, 2016
`https://panicz.github.io/pamphlet`

[40] Gopinath, K., and Hennessy, John L. *Copy Elimination in Functional Languages*. Proc. 16'th ACM POPL, Jan. 1989, 303-314.
`http://drona.csa.iisc.ernet.in/~gopi/docs/sem2c.pdf`

[41] Harrison, John, *Introduction to Functional Programming*, Cambridge University, 1997,
`http://www.cl.cam.ac.uk/teaching/Lectures/funprog-jrh-1996/all.pdf`

[42] Hederman, Lucy, *Compile Time Garbage Collection*, MS Thesis, Rice Univ. Comp. Sci. Dept., Sept. 1988.
`https://www.scss.tcd.ie/Lucy.Hederman/LHMScDissertation.pdf`

[43] Hindley, Roger J. and Felice Cardone, *History of Lambda-calculus and Combinatory Logic*, Swansea University Mathematics Department

Research Report, No. MRRS-05-06
`http://www.users.waitrose.com/~hindley/SomePapers_PDFs/2006CarHin,HistlamRp.pdf`

[44] Hoare, Charles Antony Richard and C. B. Jones (editor), *Essays in Computing Science*, Prentice Hall, 1989, ISBN 0-13-284027-8

[45] Hudak, Paul, *A Semantic Model of Reference Counting and its Abstraction*, Proc. 1986 ACM Lisp and Funct. Progr. Conf., Camb. MA, 351-363.
`https://pdfs.semanticscholar.org/5fe3/c770c0dfbb05a16fe1b969678c8ee3b1a461.pdf`

[46] Hunt, Andrew and David Thomas, *The Pragmatic Programmer: From Journeyman to Master* Pearson Education, 2000

[47] Jaffer, Aubrey, *Scheme Request for Implementation 60: Integers as Bits*, 2005
`http://srfi.schemers.org/srfi-60/`

[48] Jerram, Neil, Marius Vollmer, Martin Grabmueller, Ludovic Courtès, Andy Wingo, Aubrey Jaffer, Tom Lord, Mark Galassi, Jim Blandy, Thien-Thi Nguyen, Kevin Ryde, Mikael Djurfeldt, Christian Lynbech, Julian Graham, Gary Houston, Tim Pierce et al., *Guile Reference Manual*, Free Software Foundation, 1996-2016
`https://www.gnu.org/software/guile/manual/`

[49] Peyton Jones, Simon, Paul Hudak, John Hughes and Philip Wadler, *A History of Haskell: Being Lazy with Class*, Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), San Diego, 2007
`http://research.microsoft.com/en-us/um/people/simonpj/Papers/history-of-haskell/history.pdf`

[50] JáJá, Joseph, *An Introduction to Parallel Algorithms*, Addison-Wesley, 1992, ISBN 0-201-54856-9

[51] Keep, Andy, *Writing a Nanopass Compiler*, Clojure/Conj conference talk, 2013, video recording
`https://www.youtube.com/watch?v=Os7FE3J-U5Q`

[52] Kiselyov, Oleg, *Scheme Request For Implementation 2: AND-LET*: an AND with local bindings, a guarded LET* special form*,
`https://srfi.schemers.org/srfi-2/srfi-2.html`, 1999

[53] Knuth, Donald, *Literate Programming* CSLI Lecture Notes, no. 27, 1992, ISBN 0-937073-80-6

[54] Kowalski, Robert, *Algorithm = Logic + Control*, in *Communications of the ACM*, Volume 22 Issue 7, July 1979

[55] Kranz, David, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin, and Norman Adams, *ORBIT: An Optimizing Compiler for Scheme*, SIGPLAN '86 Proceedings of the 1986 SIGPLAN symposium on Compiler construction, Palo Alto, California, USA — June 25 - 27, 1986
`https://www.cs.purdue.edu/homes/suresh/590s-Fall2002/papers/Orbit.pdf`

[56] Kelsey, Richard and Paul Hudak, *Realistic Compilation by Program Transformation*, Yale University, 1989
`https://www.cs.purdue.edu/homes/suresh/502-Fall2008/papers/kelsey-compilation.pdf`

[57] Krishnamurthi, Shriram, *Programming Languages: Application and Interpretation*, Second Edition, November 16 2012,
`http://cs.brown.edu/~sk/Publications/Books/ProgLangs/2007-04-26/`

[58] Landin, Peter J., *The Next 700 Programming Languages*, in *Communications of the ACM*, Volume 9, Number 3, March 1966
`http://thecorememory.com/Next_700.pdf`

[59] Martin, Robert, *Clean Code: A Handbook of Agile Software Craftsmansip*, Pearson Education, 2009, ISBN 0-13-235088-2

[60] Moseley, Ben and Peter Marks, *Out of the Tar Pit*, 2006
`http://shaffner.us/cs/papers/tarpit.pdf`

[61] McCarthy, John, *Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I*, MIT, Cambridge, 1960,
`http://www-formal.stanford.edu/jmc/recursive.pdf`

[62] Moore, A. W. (ed.), *Meaning and Reference*, Oxford University Press, 1993, ISBN 0-19-875124-9

[63] Norvig, Peter, *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*, Morgan Kaufmann Publishers, San Francisco, California, 1992

[64] Okasaki, Chris, *Purely Functional Data Structures*, submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, School of Computer Science, Carnegie Mellon University, 1997
`https://www.cs.cmu.edu/~rwh/theses/okasaki.pdf`

[65] Queinnec, Christian, *Lisp in Small Pieces*, Cambridge University Press, 1996, ISBN 0-521-5466-8

[66] Kelsey, R., W. Clinger, J. Rees (editors), *Revised^5 Report on the Algorithmic Language Scheme*, in *Higher-Order and Symbolic Computation*, Vol. 11, No. 1, August, 1998 and *ACM SIGPLAN Notices*, Vol. 33, No. 9, September 1998
`http://www.schemers.org/Documents/Standards/R5RS/`

[67] Shivers, Olin, *Scheme Request for Implementation 1: List library*,
`https://srfi.schemers.org/srfi-156/`

[68] Shivers, Olin, *History of T*, 2001
`http://www.paulgraham.com/thist.html`

[69] Sitaram, Dorai, *Teach Yourself Scheme in Fixnum Days*, 1998-2003
`http://download.plt-scheme.org/doc/205/pdf/t-y-scheme.pdf`

[70] Sperber, M., Kent R. Dybvig, Matthew Flatt, Anton van Straaten (editors), *Revised^6 Report on the Algorithmic Language Scheme*, September 2007

[71] Sebesta, Robert, *Concepts of Programming Languages*, Eight Edition, Pearson Education, 2008, ISBN 978-0-321-50968-0

[72] Stańczyk, Michał Jędrzej, *On Mathematical Methods in Programming Computational Processes*, Technical University of Gdańsk, 2013
`https://github.com/drcz/drczlang`

[73] Steele, Guy Lewis Jr, and Gerald Jay Sussman, *Lambda the Ultimate Declarative*, MIT AI Lab Memo, 1976,
`http://repository.readscheme.org/ftp/papers/ai-lab-pubs/AIM-379.pdf`

[74] Stepanov, Alexander, *Notes on Higher Order Programming in Scheme*, August 1986
`http://stepanovpapers.com/schemenotes/notes.pdf`

[75] Stroustrup, Bjarne, *The C++ Programming Language*, Third Edition, AT&T Labs, Murray Hill, New Jersey

[76] Szabó, Zoltán Gendler, *Compositionality*, in Edward N. Zalta (ed.) *The Stanford Encyclopedia of Philosophy*, Fall 2013 Edition,
`http://plato.stanford.edu/archives/fall2013/entries/compositionality/`

[77] Sussman, Gerald Jay and Jack Wisdom, *Structure adn Interpretation of Classical Mechanics* MIT Press, 2001, ISBN 0-262-019455-4
`https://mitpress.mit.edu/sites/default/files/titles/content/sicm/book.html`

[78] van Heijenoort, Jean, *From Frege to Gödel: A Source Book in Mathematical Logic, 1879-1931*, Harvard University Press, 1967

[79] Victor, Bret, *The Future of Programming*, video lecture
`https://www.youtube.com/watch?v=8pTEmbeENF4`

[80] Wadler, Philip, *The essence of functional programming*,
`https://cs.uwaterloo.ca/~david/cs442/monads.pdf`

[81] Wadler, Philip, *Monads for functional programming*, in J. Jeuring and E. Meijer (editors) *Advanced Functional Programming*, Proceedings of the Båstad Spring School, May 1995, Springer Verlag Lecture Notes in Computer Science 925.
`http://homepages.inf.ed.ac.uk/wadler/papers/marktoberdorf/baastad.pdf`

[82] Wirth, Niklaus, *Algorithms + Data Structures = Programs*, Prentice Hall, 1976, ISBN 978-0-13-022418-7

[83] Wright, Andrew K., and Robert Cartwright, *A Practical Soft Type System for Scheme* in *ACM Transactions on Programming Languages and Systems*, Vol. 19 No. 1, January 1997, Pages 87-152
`http://www.iro.umontreal.ca/~feeley/cours/ift6232/doc/pres2/practical-soft-type-system-for-scheme.pdf`

[84] Wikipedia, *Register allocation*,
`https://en.wikipedia.org/wiki/Register_allocation`
access 21/04/2017