Essentials of Compilation

An Incremental Approach

Jeremy G. Siek
Indiana University

with contributions from:
Carl Factora
Andre Kuhlenschmidt
Ryan R. Newton
Ryan Scott
Cameron Swords
Michael M. Vitousek
Michael Vollmer

October 23, 2020

This book is dedicated to the programming language wonks at Indiana University.

Contents

List of Figures

Preface

The tradition of compiler writing at Indiana University goes back to research and courses about programming languages by Daniel Friedman in the 1970's and 1980's. Dan conducted research on lazy evaluation [Friedman and Wise, 1976] in the context of Lisp [McCarthy, 1960] and then studied continuations [Felleisen and Friedman, 1986] and macros [Kohlbecker et al., 1986] in the context of the Scheme [Sussman and Jr., 1975], a dialect of Lisp. One of the students of those courses, Kent Dybvig, went on to build Chez Scheme [Dybvig, 2006], a production-quality and efficient compiler for Scheme. After completing his Ph.D. at the University of North Carolina, Kent returned to teach at Indiana University. Throughout the 1990's and 2000's, Kent continued development of Chez Scheme and taught the compiler course.

The compiler course evolved to incorporate novel pedagogical ideas while also including elements of effective real-world compilers. One of Dan's ideas was to split the compiler into many small "passes" so that the code for each pass would be easy to understood in isolation. (In contrast, most compilers of the time were organized into only a few monolithic passes for reasons of compile-time efficiency.) Kent, with later help from his students Dipanwita Sarkar and Andrew Keep, developed infrastructure to support this approach and evolved the course, first to use micro-sized passes and then into even smaller nano passes [Sarkar et al., 2004, Keep, 2012]. Jeremy Siek was a student in this compiler course in the early 2000's, as part of his Ph.D. studies at Indiana University. Needless to say, Jeremy enjoyed the course immensely!

During that time, another student named Abdulaziz Ghuloum observed that the front-to-back organization of the course made it difficult for students to understand the rationale for the compiler design. Abdulaziz proposed an incremental approach in which the students build the compiler in stages; they start by implementing a complete compiler for a very small subset of the input language and in each subsequent stage they add a language feature

and add or modify passes to handle the new feature [Ghuloum, 2006]. In this way, the students see how the language features motivate aspects of the compiler design.

After graduating from Indiana University in 2005, Jeremy went on to teach at the University of Colorado. He adapted the nano pass and incremental approaches to compiling a subset of the Python language [Siek and Chang, 2012]. Python and Scheme are quite different on the surface but there is a large overlap in the compiler techniques required for the two languages. Thus, Jeremy was able to teach much of the same content from the Indiana compiler course. He very much enjoyed teaching the course organized in this way, and even better, many of the students learned a lot and got excited about compilers.

Jeremy returned to teach at Indiana University in 2013. In his absence the compiler course had switched from the front-to-back organization to a back-to-front organization. Seeing how well the incremental approach worked at Colorado, he started porting and adapting the structure of the Colorado course back into the land of Scheme. In the meantime Indiana had moved on from Scheme to Racket, so the course is now about compiling a subset of Racket (and Typed Racket) to the x86 assembly language. The compiler is implemented in Racket 7.1 [Flatt and PLT, 2014].

This is the textbook for the incremental version of the compiler course at Indiana University (Spring 2016 - present) and it is the first open textbook for an Indiana compiler course. With this book we hope to make the Indiana compiler course available to people that have not had the chance to study in Bloomington in person. Many of the compiler design decisions in this book are drawn from the assignment descriptions of Dybvig and Keep [2010]. We have captured what we think are the most important topics from Dybvig and Keep [2010] but we have omitted topics that we think are less interesting conceptually and we have made simplifications to reduce complexity. In this way, this book leans more towards pedagogy than towards the efficiency of the generated code. Also, the book differs in places where we saw the opportunity to make the topics more fun, such as in relating register allocation to Sudoku (Chapter 3).

Prerequisites

The material in this book is challenging but rewarding. It is meant to prepare students for a lifelong career in programming languages.

The book uses the Racket language both for the implementation of the

compiler and for the language that is compiled, so a student should be proficient with Racket (or Scheme) prior to reading this book. There are many excellent resources for learning Scheme and Racket [Dybvig, 1987, Abelson and Sussman, 1996, Friedman and Felleisen, 1996, Felleisen et al., 2001, 2013, Flatt et al., 2014]. It is helpful but not necessary for the student to have prior exposure to the x86 (or x86-64) assembly language [Intel, 2015], as one might obtain from a computer systems course [Bryant and O'Hallaron, 2005, 2010]. This book introduces the parts of x86-64 assembly language that are needed.

Acknowledgments

Many people have contributed to the ideas, techniques, organization, and teaching of the materials in this book. We especially thank the following people.

- Bor-Yuh Evan Chang
- Kent Dybvig
- Daniel P. Friedman
- Ronald Garcia
- Abdulaziz Ghuloum
- Jay McCarthy
- Dipanwita Sarkar
- Andrew Keep
- Oscar Waddell
- Michael Wollowski

Jeremy G. Siek
`http://homes.soic.indiana.edu/jsiek`

1

Preliminaries

In this chapter we review the basic tools that are needed to implement a compiler. Programs are typically input by a programmer as text, i.e., a sequence of characters. The program-as-text representation is called *concrete syntax*. We use concrete syntax to concisely write down and talk about programs. Inside the compiler, we use *abstract syntax trees* (ASTs) to represent programs in a way that efficiently supports the operations that the compiler needs to perform. The translation from concrete syntax to abstract syntax is a process called *parsing* Aho et al. [1986]. We do not cover the theory and implementation of parsing in this book. A parser is provided in the supporting materials for translating from concrete syntax to abstract syntax for the languages used in this book.

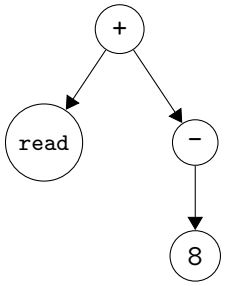
ASTs can be represented in many different ways inside the compiler, depending on the programming language used to write the compiler. We use Racket's `struct` feature to represent ASTs (Section 1.1). We use grammars to define the abstract syntax of programming languages (Section 1.2) and pattern matching to inspect individual nodes in an AST (Section 1.3). We use recursion to construct and deconstruct entire ASTs (Section 1.4). This chapter provides an brief introduction to these ideas.

1.1 Abstract Syntax Trees and Racket Structures

Compilers use abstract syntax trees to represent programs because compilers often need to ask questions like: for a given part of a program, what kind of language feature is it? What are the sub-parts of this part of the program? Consider the program on the left and its AST on the right. This program is an addition and it has two sub-parts, a read operation and a negation. The

negation has another sub-part, the integer constant `8`. By using a tree to represent the program, we can easily follow the links to go from one part of a program to its sub-parts.

```
(+ (read) (- 8))
```



(1.1)

We use the standard terminology for trees to describe ASTs: each circle above is called a *node*. The arrows connect a node to its *children* (which are also nodes). The top-most node is the *root*. Every node except for the root has a *parent* (the node it is the child of). If a node has no children, it is a *leaf* node. Otherwise it is an *internal* node.

We define a Racket `struct` for each kind of node. For this chapter we require just two kinds of nodes: one for integer constants and one for primitive operations. The following is the `struct` definition for integer constants.

```
(struct Int (value))
```

An integer node includes just one thing: the integer value. To create a AST node for the integer 8, we write `(Int 8)`.

```
(define eight (Int 8))
```

We say that the value created by `(Int 8)` is an *instance* of the `Int` structure.

The following is the `struct` definition for primitives operations.

```
(struct Prim (op arg*))
```

A primitive operation node includes an operator symbol `op` and a list of children `arg*`. For example, to create an AST that negates the number 8, we write `(Prim '- (list eight))`.

```
(define neg-eight (Prim '- (list eight)))
```

Primitive operations may have zero or more children. The `read` operator has zero children:

```
(define rd (Prim 'read '()))
```

whereas the addition operator has two children:

```
(define ast1.1 (Prim '+ (list rd neg-eight)))
```

We have made a design choice regarding the `Prim` structure. Instead of using one structure for many different operations (`read`, `+`, and `-`), we could have instead defined a structure for each operation, as follows.

```
(struct Read ())
(struct Add (left right))
(struct Neg (value))
```

The reason we choose to use just one structure is that in many parts of the compiler the code for the different primitive operators is the same, so we might as well just write that code once, which is enabled by using a single structure.

When compiling a program such as (1.1), we need to know that the operation associated with the root node is addition and we need to be able to access its two children. Racket provides pattern matching over structures to support these kinds of queries, as we shall see in Section 1.3.

In this book, we often write down the concrete syntax of a program even when we really have in mind the AST because the concrete syntax is more concise. We recommend that, in your mind, you always think of programs as abstract syntax trees.

1.2 Grammars

A programming language can be thought of as a *set* of programs. The set is typically infinite (one can always create larger and larger programs), so one cannot simply describe a language by listing all of the programs in the language. Instead we write down a set of rules, a *grammar*, for building programs. Grammars are often used to define the concrete syntax of a language, but they can also be used to describe the abstract syntax. We shall write our rules in a variant of Backus-Naur Form (BNF) [Backus et al., 1960, Knuth, 1964]. As an example, we describe a small language, named R_0, that consists of integers and arithmetic operations.

The first grammar rule for the abstract syntax of R_0 says that an instance of the `Int` structure is an expression:

$$exp ::= \texttt{(Int } int\texttt{)} \tag{1.2}$$

Each rule has a left-hand-side and a right-hand-side. The way to read a rule is that if you have all the program parts on the right-hand-side, then

you can create an AST node and categorize it according to the left-hand-side. A name such as *exp* that is defined by the grammar rules is a *non-terminal*. The name *int* is a also a non-terminal, but instead of defining it with a grammar rule, we define it with the following explanation. We make the simplifying design decision that all of the languages in this book only handle machine-representable integers. On most modern machines this corresponds to integers represented with 64-bits, i.e., the in range -2^{63} to $2^{63}-1$. We restrict this range further to match the Racket `fixnum` datatype, which allows 63-bit integers on a 64-bit machine. So an *int* is a sequence of decimals (0 to 9), possibly starting with $-$ (for negative integers), such that the sequence of decimals represent an integer in range -2^{62} to $2^{62}-1$.

The second grammar rule is the `read` operation that receives an input integer from the user of the program.

$$\textit{exp} ::= \texttt{(Prim 'read '())} \tag{1.3}$$

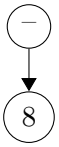
The third rule says that, given an *exp* node, you can build another *exp* node by negating it.

$$\textit{exp} ::= \texttt{(Prim '- (list } \textit{exp} \texttt{))} \tag{1.4}$$

Symbols in typewriter font such as `-` and `read` are *terminal* symbols and must literally appear in the program for the rule to be applicable.

We can apply the rules to build ASTs in the R_0 language. For example, by rule (1.2), `(Int 8)` is an *exp*, then by rule (1.4), the following AST is an *exp*.

```
(Prim '- (list (Int 8)))
```



(1.5)

The next grammar rule defines addition expressions:

$$\textit{exp} ::= \texttt{(Prim '+ (list } \textit{exp}\ \textit{exp}\texttt{))} \tag{1.6}$$

We can now justify that the AST (1.1) is an *exp* in R_0. We know that `(Prim 'read '())` is an *exp* by rule (1.3) and we have already shown that `(Prim '- (list (Int 8)))` is an *exp*, so we apply rule (1.6) to show that

```
(Prim '+ (list (Prim 'read '()) (Prim '- (list (Int 8)))))
```

$$\begin{array}{rcl} \mathit{exp} & ::= & \mathit{int} \mid (\texttt{read}) \mid (\texttt{-}\ \mathit{exp}) \mid (\texttt{+}\ \mathit{exp}\ \mathit{exp}) \\ R_0 & ::= & \mathit{exp} \end{array}$$

Figure 1.1: The concrete syntax of R_0.

$$\begin{array}{rcl} \mathit{exp} & ::= & (\texttt{Int}\ \mathit{int}) \mid (\texttt{Prim 'read '()}) \mid (\texttt{Prim '-}\ (\texttt{list}\ \mathit{exp}\ )) \\ & \mid & (\texttt{Prim '+}\ (\texttt{list}\ \mathit{exp}\ \mathit{exp})) \\ R_0 & ::= & (\texttt{Program '()}\ \mathit{exp}) \end{array}$$

Figure 1.2: The abstract syntax of R_0.

is an *exp* in the R_0 language.

If you have an AST for which the above rules do not apply, then the AST is not in R_0. For example, the program `(- (read) (+ 8))` is not in R_0 because there are no rules for `+` with only one argument, nor for `-` with two arguments. Whenever we define a language with a grammar, the language only includes those programs that are justified by the rules.

The last grammar rule for R_0 states that there is a `Program` node to mark the top of the whole program:

$$R_0 ::= (\texttt{Program '()}\ \mathit{exp})$$

The `Program` structure is defined as follows

```
(struct Program (info body))
```

where `body` is an expression. In later chapters, the `info` part will be used to store auxiliary information but for now it is just the empty list.

It is common to have many grammar rules with the same left-hand side but different right-hand sides, such as the rules for *exp* in the grammar of R_0. As a short-hand, a vertical bar can be used to combine several right-hand-sides into a single rule.

We collect all of the grammar rules for the abstract syntax of R_0 in Figure 1.2. The concrete syntax for R_0 is defined in Figure 1.1.

The `read-program` function provided in `utilities.rkt` of the support materials reads a program in from a file (the sequence of characters in the concrete syntax of Racket) and parses it into an abstract syntax tree. See the description of `read-program` in Appendix 12.2 for more details.

1.3 Pattern Matching

As mentioned in Section 1.1, compilers often need to access the parts of an AST node. Racket provides the `match` form to access the parts of a structure. Consider the following example and the output on the right.

```
(match ast1.1                                  |
  [(Prim op (list child1 child2))              |
    (print op)])                               |  '+
```

In the above example, the `match` form takes the AST (1.1) and binds its parts to the three pattern variables `op`, `child1`, and `child2`. In general, a match clause consists of a *pattern* and a *body*. Patterns are recursively defined to be either a pattern variable, a structure name followed by a pattern for each of the structure's arguments, or an S-expression (symbols, lists, etc.). (See Chapter 12 of The Racket Guide[1] and Chapter 9 of The Racket Reference[2] for a complete description of `match`.) The body of a match clause may contain arbitrary Racket code. The pattern variables can be used in the scope of the body.

A `match` form may contain several clauses, as in the following function `leaf?` that recognizes when an R_0 node is a leaf. The `match` proceeds through the clauses in order, checking whether the pattern can match the input AST. The body of the first clause that matches is executed. The output of `leaf?` for several ASTs is shown on the right.

```
(define (leaf? arith)                          |
  (match arith                                 |
    [(Int n) #t]                               |
    [(Prim 'read '()) #t]                      |
    [(Prim '- (list c1)) #f]                   |
    [(Prim '+ (list c1 c2)) #f]))              |
                                               |
(leaf? (Prim 'read '()))                       |  #t
(leaf? (Prim '- (list (Int 8))))               |  #f
(leaf? (Int 8))                                |  #t
```

When writing a `match`, we refer to the grammar definition to identify which non-terminal we are expecting to match against, then we make sure that 1) we have one clause for each alternative of that non-terminal and 2)

[1] `https://docs.racket-lang.org/guide/match.html`
[2] `https://docs.racket-lang.org/reference/match.html`

that the pattern in each clause corresponds to the corresponding right-hand side of a grammar rule. For the `match` in the `leaf?` function, we refer to the grammar for R_0 in Figure 1.2. The *exp* non-terminal has 4 alternatives, so the `match` has 4 clauses. The pattern in each clause corresponds to the right-hand side of a grammar rule. For example, the pattern `(Prim '+ (list c1 c2))` corresponds to the right-hand side `(Prim '+ (list` *exp exp*`))`. When translating from grammars to patterns, replace non-terminals such as *exp* with pattern variables of your choice (e.g. `c1` and `c2`).

1.4 Recursion

Programs are inherently recursive. For example, an R_0 expression is often made of smaller expressions. Thus, the natural way to process an entire program is with a recursive function. As a first example of such a recursive function, we define `exp?` below, which takes an arbitrary value and determines whether or not it is an R_0 expression. When a recursive function is defined using a sequence of match clauses that correspond to a grammar, and the body of each clause makes a recursive call on each child node, then we say the function is defined by *structural recursion*[3]. Below we also define a second function, named `R0?`, that determines whether a value is an R_0 program. In general we can expect to write one recursive function to handle each non-terminal in a grammar.

[3]This principle of structuring code according to the data definition is advocated in the book *How to Design Programs* `http://www.ccs.neu.edu/home/matthias/HtDP2e/`.

```
(define (exp? ast)
  (match ast
    [(Int n) #t]
    [(Prim 'read '()) #t]
    [(Prim '- (list e)) (exp? e)]
    [(Prim '+ (list e1 e2))
     (and (exp? e1) (exp? e2))]
    [else #f]))

(define (R0? ast)
  (match ast
    [(Program '() e) (exp? e)]
    [else #f]))

(R0? (Program '() ast1.1))
(R0? (Program '()
      (Prim '- (list (Prim 'read '())
                     (Prim '+ (list (Num 8)))))))
```

```
#t
#f
```

You may be tempted to merge the two functions into one, like this:

```
(define (R0? ast)
  (match ast
    [(Int n) #t]
    [(Prim 'read '()) #t]
    [(Prim '- (list e)) (R0? e)]
    [(Prim '+ (list e1 e2)) (and (R0? e1) (R0? e2))]
    [(Program '() e) (R0? e)]
    [else #f]))
```

Sometimes such a trick will save a few lines of code, especially when it comes to the `Program` wrapper. Yet this style is generally *not* recommended because it can get you into trouble. For example, the above function is subtly wrong: `(R0? (Program '() (Program '() (Int 3))))` will return true, when it should return false.

1.5 Interpreters

The meaning, or semantics, of a program is typically defined in the specification of the language. For example, the Scheme language is defined in the report by Sperber et al. [2009]. The Racket language is defined in its reference manual [Flatt and PLT, 2014]. In this book we use an interpreter

```
(define (interp-exp e)
  (match e
    [(Int n) n]
    [(Prim 'read '())
     (define r (read))
     (cond [(fixnum? r) r]
           [else (error 'interp-R0 "expected an integer" r)])]
    [(Prim '- (list e))
     (define v (interp-exp e))
     (fx- 0 v)]
    [(Prim '+ (list e1 e2))
     (define v1 (interp-exp e1))
     (define v2 (interp-exp e2))
     (fx+ v1 v2)]
    ))

(define (interp-R0 p)
  (match p
    [(Program '() e) (interp-exp e)]
    ))
```

Figure 1.3: Interpreter for the R_0 language.

to define the meaning of each language that we consider, following Reynolds' advice [Reynolds, 1972]. An interpreter that is designated (by some people) as the definition of a language is called a *definitional interpreter*. We warm up by creating a definitional interpreter for the R_0 language, which serves as a second example of structural recursion. The `interp-R0` function is defined in Figure 1.3. The body of the function is a match on the input program followed by a call to the `interp-exp` helper function, which in turn has one match clause per grammar rule for R_0 expressions.

Let us consider the result of interpreting a few R_0 programs. The following program adds two integers.

```
(+ 10 32)
```

The result is `42`. We wrote the above program in concrete syntax, whereas the parsed abstract syntax is:

```
(Program '() (Prim '+ (list (Int 10) (Int 32))))
```

The next example demonstrates that expressions may be nested within each other, in this case nesting several additions and negations.

```
(+ 10 (- (+ 12 20)))
```

What is the result of the above program?

As mentioned previously, the R_0 language does not support arbitrarily-large integers, but only 63-bit integers, so we interpret the arithmetic operations of R_0 using fixnum arithmetic in Racket. Suppose

$$n = 999999999999999999$$

which indeed fits in 63-bits. What happens when we run the following program in our interpreter?

```
(+ (+ (+ n n) (+ n n)) (+ (+ n n) (+ n n)))))
```

It produces an error:

```
fx+: result is not a fixnum
```

We establish the convention that if running the definitional interpreter on a program produces an error, then the meaning of that program is *unspecified*. That means a compiler for the language is under no obligations regarding that program; it may or may not produce an executable, and if it does, that executable can do anything. This convention applies to the languages defined in this book, as a way to simplify the student's task of implementing them, but this convention is not applicable to all programming languages.

Moving on to the last feature of the R_0 language, the `read` operation prompts the user of the program for an integer. Recall that program (1.1) performs a `read` and then subtracts `8`. So if we run

```
(interp-R0 (Program '() ast1.1))
```

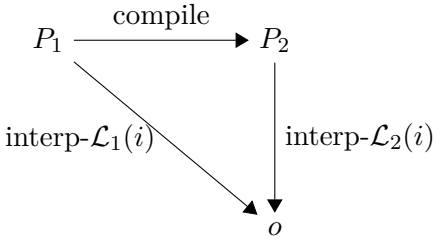
and if the input is `50`, then we get the answer to life, the universe, and everything: `42`![4]

We include the `read` operation in R_0 so a clever student cannot implement a compiler for R_0 that simply runs the interpreter during compilation to obtain the output and then generates the trivial code to produce the output. (Yes, a clever student did this in the first instance of this course.)

The job of a compiler is to translate a program in one language into a program in another language so that the output program behaves the same way as the input program does according to its definitional interpreter. This idea is depicted in the following diagram. Suppose we have two languages, $\mathcal{L}_1$ and $\mathcal{L}_2$, and an interpreter for each language. Suppose that the compiler translates program P_1 in language $\mathcal{L}_1$ into program P_2 in language $\mathcal{L}_2$. Then

[4] *The Hitchhiker's Guide to the Galaxy* by Douglas Adams.

interpreting P_1 and P_2 on their respective interpreters with input i should yield the same output o.



(1.7)

In the next section we see our first example of a compiler.

1.6 Example Compiler: a Partial Evaluator

In this section we consider a compiler that translates R_0 programs into R_0 programs that may be more efficient, that is, this compiler is an optimizer. This optimizer eagerly computes the parts of the program that do not depend on any inputs, a process known as *partial evaluation* Jones et al. [1993]. For example, given the following program

```
(+ (read) (- (+ 5 3)))
```

our compiler will translate it into the program

```
(+ (read) -8)
```

Figure 1.4 gives the code for a simple partial evaluator for the R_0 language. The output of the partial evaluator is an R_0 program. In Figure 1.4, the structural recursion over *exp* is captured in the `pe-exp` function whereas the code for partially evaluating the negation and addition operations is factored into two separate helper functions: `pe-neg` and `pe-add`. The input to these helper functions is the output of partially evaluating the children.

The `pe-neg` and `pe-add` functions check whether their arguments are integers and if they are, perform the appropriate arithmetic. Otherwise, they create an AST node for the operation (either negation or addition).

To gain some confidence that the partial evaluator is correct, we can test whether it produces programs that get the same result as the input programs. That is, we can test whether it satisfies Diagram (1.7). The following code runs the partial evaluator on several examples and tests the output program. The `parse-program` and `assert` functions are defined in

```
(define (pe-neg r)
  (match r
    [(Int n) (Int (fx- 0 n))]
    [else (Prim '- (list r))]))

(define (pe-add r1 r2)
  (match* (r1 r2)
    [((Int n1) (Int n2)) (Int (fx+ n1 n2))]
    [(_ _) (Prim '+ (list r1 r2))]))

(define (pe-exp e)
  (match e
    [(Int n) (Int n)]
    [(Prim 'read '()) (Prim 'read '())]
    [(Prim '- (list e1)) (pe-neg (pe-exp e1))]
    [(Prim '+ (list e1 e2)) (pe-add (pe-exp e1) (pe-exp e2))]
    ))

(define (pe-R0 p)
  (match p
    [(Program '() e) (Program '() (pe-exp e))]
    ))
```

Figure 1.4: A partial evaluator for R_0 expressions.

Appendix 12.2.

```
(define (test-pe p)
  (assert "testing pe-R0"
     (equal? (interp-R0 p) (interp-R0 (pe-R0 p)))))

(test-pe (parse-program `(program () (+ 10 (- (+ 5 3))))))
(test-pe (parse-program `(program () (+ 1 (+ 3 1)))))
(test-pe (parse-program `(program () (- (+ 3 (- 5))))))
```

2

Integers and Variables

This chapter is about compiling the subset of Racket that includes integer arithmetic and local variable binding, which we name R_1, to x86-64 assembly code [Intel, 2015]. Henceforth we shall refer to x86-64 simply as x86. The chapter begins with a description of the R_1 language (Section 2.1) followed by a description of x86 (Section 2.2). The x86 assembly language is large, so we discuss only what is needed for compiling R_1. We introduce more of x86 in later chapters. Once we have introduced R_1 and x86, we reflect on their differences and come up with a plan to break down the translation from R_1 to x86 into a handful of steps (Section 2.3). The rest of the sections in this chapter give detailed hints regarding each step (Sections 2.4 through 2.9). We hope to give enough hints that the well-prepared reader, together with a few friends, can implement a compiler from R_1 to x86 in a couple weeks while at the same time leaving room for some fun and creativity. To give the reader a feeling for the scale of this first compiler, the instructor solution for the R_1 compiler is less than 500 lines of code.

2.1 The R_1 Language

The R_1 language extends the R_0 language with variable definitions. The concrete syntax of the R_1 language is defined by the grammar in Figure 2.1 and the abstract syntax is defined in Figure 2.2. The non-terminal *var* may be any Racket identifier. As in R_0, `read` is a nullary operator, `-` is a unary operator, and `+` is a binary operator. Similar to R_0, the abstract syntax of R_1 includes the `Program` struct to mark the top of the program. Despite the simplicity of the R_1 language, it is rich enough to exhibit several compilation techniques.

exp	::=	*int* | (`read`) | (`-` *exp*) | (`+` *exp exp*)
	|	*var* | (`let` ([*var exp*]) *exp*)
R_1	::=	*exp*

Figure 2.1: The concrete syntax of R_1.

exp	::=	(`Int` *int*) | (`Prim 'read '()`)
	|	(`Prim '-` (`list` *exp*)) | (`Prim '+` (`list` *exp exp*))
	|	(`Var` *var*) | (`Let` *var exp exp*)
R_1	::=	(`Program '()` *exp*)

Figure 2.2: The abstract syntax of R_1.

Let us dive further into the syntax and semantics of the R_1 language. The `Let` feature defines a variable for use within its body and initializes the variable with the value of an expression. The abstract syntax for `Let` is defined in Figure 2.2. The concrete syntax for `Let` is

(`let` ([*var exp*]) *exp*)

For example, the following program initializes `x` to 32 and then evaluates the body `(+ 10 x)`, producing 42.

```
(let ([x (+ 12 20)]) (+ 10 x))
```

When there are multiple `let`'s for the same variable, the closest enclosing `let` is used. That is, variable definitions overshadow prior definitions. Consider the following program with two `let`'s that define variables named `x`. Can you figure out the result?

```
(let ([x 32]) (+ (let ([x 10]) x) x))
```

For the purposes of depicting which variable uses correspond to which definitions, the following shows the `x`'s annotated with subscripts to distinguish them. Double check that your answer for the above is the same as your answer for this annotated version of the program.

```
(let ([x1 32]) (+ (let ([x2 10]) x2) x1))
```

The initializing expression is always evaluated before the body of the `let`, so in the following, the `read` for `x` is performed before the `read` for `y`. Given the input 52 then 10, the following produces 42 (not -42).

```
(let ([x (read)]) (let ([y (read)]) (+ x (- y))))
```

Figure 2.3 shows the definitional interpreter for the R_1 language. It extends the interpreter for R_0 with two new `match` clauses for variables and for `let`. For `let`, we need a way to communicate the value of a variable to all the uses of a variable. To accomplish this, we maintain a mapping from variables to values. Throughout the compiler we often need to map variables to information about them. We refer to these mappings as *environments* [1]. For simplicity, we use an association list (alist) to represent the environment. The sidebar to the right gives a brief introduction to alists and the `racket/dict` package. The `interp-R1` function takes the current environment, `env`, as an extra parameter. When the interpreter encounters a variable, it finds the corresponding value using the `dict-ref` function. When the interpreter encounters a `Let`, it evaluates the initializing expression, extends the environment with the result value bound to the variable, using `dict-set`, then evaluates the body of the `Let`.

Association Lists as Dictionaries

An *association list* (alist) is a list of key-value pairs. For example, we can map people to their ages with an alist.

```
(define ages
  '((jane . 25) (sam . 24) (kate . 45)))
```

The *dictionary* interface is for mapping keys to values. Every alist implements this interface. The package `racket/dict` provides many functions for working with dictionaries. Here are a few of them:

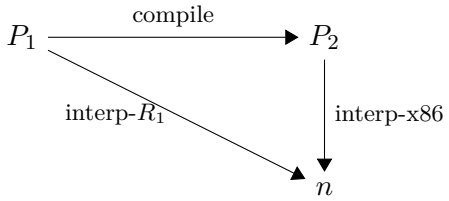
`(dict-ref` *dict key*`)` returns the value associated with the given *key*.

`(dict-set` *dict key val*`)` returns a new dictionary that maps *key* to *val* but otherwise is the same as *dict*.

`(in-dict` *dict*`)` returns the sequence of keys and values in *dict*. For example, the following creates a new alist in which the ages are incremented.

```
(for/list ([(k v) (in-dict ages)])
  (cons k (add1 v)))
```

The goal for this chapter is to implement a compiler that translates any program P_1 written in the R_1 language into an x86 assembly program P_2 such that P_2 exhibits the same behavior when run on a computer as the P_1 program interpreted by `interp-R1`. That is, they both output the same integer n. We depict this correctness criteria in the following diagram.



In the next section we introduce enough of the x86 assembly language to compile R_1.

[1] Another common term for environment in the compiler literature is *symbol table*.

```
(define (interp-exp env)
  (lambda (e)
    (match e
      [(Int n) n]
      [(Prim 'read '())
       (define r (read))
       (cond [(fixnum? r) r]
             [else (error 'interp-R1 "expected an integer" r)])]
      [(Prim '- (list e))
       (define v ((interp-exp env) e))
       (fx- 0 v)]
      [(Prim '+ (list e1 e2))
       (define v1 ((interp-exp env) e1))
       (define v2 ((interp-exp env) e2))
       (fx+ v1 v2)]
      [(Var x) (dict-ref env x)]
      [(Let x e body)
       (define new-env (dict-set env x ((interp-exp env) e)))
       ((interp-exp new-env) body)]
      )))

(define (interp-R1 p)
  (match p
    [(Program '() e) ((interp-exp '()) e)]
    ))
```

Figure 2.3: Interpreter for the R_1 language.

reg	::=	`rsp` | `rbp` | `rax` | `rbx` | `rcx` | `rdx` | `rsi` | `rdi` | `r8` | `r9` | `r10` | `r11` | `r12` | `r13` | `r14` | `r15`
arg	::=	`$`*int* | `%`*reg* | *int*`(%`*reg*`)`
instr	::=	`addq` *arg*`,`*arg* | `subq` *arg*`,`*arg* | `negq` *arg* | `movq` *arg*`,`*arg* | `callq` *label* | `pushq` *arg* | `popq` *arg* | `retq` | `jmp` *label* *label*`:` *instr*
$x86_0$	::=	`.globl main` `main:` *instr* ...

Figure 2.4: The concrete syntax of the x86$_0$ assembly language (AT&T syntax).

2.2 The x86$_0$ Assembly Language

Figure 2.4 defines the concrete syntax for the subset of the x86 assembly language needed for this chapter, which we call x86$_0$. An x86 program begins with a `main` label followed by a sequence of instructions. In the grammar, elipses such as ... are used to indicate a sequence of items, e.g., *instr* ... is a sequence of instructions. An x86 program is stored in the computer's memory and the computer has a *program counter* (PC) that points to the address of the next instruction to be executed. For most instructions, once the instruction is executed, the program counter is incremented to point to the immediately following instruction in memory. Most x86 instructions take two operands, where each operand is either an integer constant (called *immediate value*), a *register*, or a memory location. A register is a special kind of variable. Each one holds a 64-bit value; there are 16 registers in the computer and their names are given in Figure 2.4. The computer's memory as a mapping of 64-bit addresses to 64-bit values[2]. We use the AT&T syntax expected by the GNU assembler, which comes with the `gcc` compiler that we use for compiling assembly code to machine code. Appendix 12.3 is a quick-reference for all of the x86 instructions used in this book.

An immediate value is written using the notation `$`n where n is an integer. A register is written with a `%` followed by the register name, such as `%rax`. An access to memory is specified using the syntax n(`%`r), which obtains the address stored in register r and then adds n bytes to the address.

[2]This simple story suffices for describing how sequential programs access memory but is not sufficient for multi-threaded programs. However, multi-threaded execution is beyond the scope of this book.

```
        .globl main
main:
        movq   $10, %rax
        addq   $32, %rax
        retq
```

Figure 2.5: An x86 program equivalent to `(+ 10 32)`.

The resulting address is used to either load or store to memory depending on whether it occurs as a source or destination argument of an instruction.

An arithmetic instruction such as `addq` s`,` d reads from the source s and destination d, applies the arithmetic operation, then writes the result back to the destination d. The move instruction `movq` s`,` d reads from s and stores the result in d. The `callq` *label* instruction executes the procedure specified by the label and `retq` returns from a procedure to its caller. We discuss procedure calls in more detail later in this chapter and in Chapter 6. The `jmp` *label* instruction updates the program counter to the address of the instruction after the specified label.

Figure 2.5 depicts an x86 program that is equivalent to `(+ 10 32)`. The `globl` directive says that the `main` procedure is externally visible, which is necessary so that the operating system can call it. The label `main:` indicates the beginning of the `main` procedure which is where the operating system starts executing this program. The instruction `movq $10, %rax` puts 10 into register `rax`. The following instruction `addq $32, %rax` adds 32 to the 10 in `rax` and puts the result, 42, back into `rax`. The last instruction, `retq`, finishes the `main` function by returning the integer in `rax` to the operating system. The operating system interprets this integer as the program's exit code. By convention, an exit code of 0 indicates that a program completed successfully, and all other exit codes indicate various errors. Nevertheless, we return the result of the program as the exit code.

Unfortunately, x86 varies in a couple ways depending on what operating system it is assembled in. The code examples shown here are correct on Linux and most Unix-like platforms, but when assembled on Mac OS X, labels like `main` must be prefixed with an underscore, as in `_main`.

We exhibit the use of memory for storing intermediate results in the next example. Figure 2.6 lists an x86 program that is equivalent to `(+ 52 (- 10))`. This program uses a region of memory called the *procedure call stack* (or *stack* for short). The stack consists of a separate *frame* for each procedure call. The memory layout for an individual frame is shown in

```
start:
       movq   $10, -8(%rbp)
       negq   -8(%rbp)
       movq   -8(%rbp), %rax
       addq   $52, %rax
       jmp conclusion

       .globl main
main:
       pushq  %rbp
       movq   %rsp, %rbp
       subq   $16, %rsp
       jmp start
conclusion:
       addq   $16, %rsp
       popq   %rbp
       retq
```

Figure 2.6: An x86 program equivalent to `(+ 10 32)`.

Position	Contents
8(`%rbp`)	return address
0(`%rbp`)	old `rbp`
-8(`%rbp`)	variable 1
-16(`%rbp`)	variable 2
...	...
0(`%rsp`)	variable n

Figure 2.7: Memory layout of a frame.

Figure 2.7. The register `rsp` is called the *stack pointer* and points to the item at the top of the stack. The stack grows downward in memory, so we increase the size of the stack by subtracting from the stack pointer. In the context of a procedure call, the *return address* is the next instruction after the call instruction on the caller side. During a function call, the return address is pushed onto the stack. The register `rbp` is the *base pointer* and is used to access variables associated with the current procedure call. The base pointer of the caller is pushed onto the stack after the return address. We number the variables from 1 to n. Variable 1 is stored at address -8(`%rbp`), variable 2 at -16(`%rbp`), etc.

Getting back to the program in Figure 2.6, consider how control is trans-

fered from the operating system to the `main` function. The operating system issues a `callq main` instruction which pushes its return address on the stack and then jumps to `main`. In x86-64, the stack pointer `rsp` must be divisible by 16 bytes prior to the execution of any `callq` instruction, so when control arrives at `main`, the `rsp` is 8 bytes out of alignment (because the `callq` pushed the return address). The first three instructions are the typical *prelude* for a procedure. The instruction `pushq %rbp` saves the base pointer for the caller onto the stack and subtracts 8 from the stack pointer. At this point the stack pointer is back to being 16-byte aligned. The second instruction `movq %rsp, %rbp` changes the base pointer so that it points the location of the old base pointer. The instruction `subq $16, %rsp` moves the stack pointer down to make enough room for storing variables. This program needs one variable (8 bytes) but we round up to 16 bytes to maintain the 16-byte alignment of the `rsp`. With the `rsp` aligned, we are ready to make calls to other functions. The last instruction of the prelude is `jmp start`, which transfers control to the instructions that were generated from the Racket expression `(+ 10 32)`.

The four instructions under the label `start` carry out the work of computing `(+ 52 (- 10))`. The first instruction `movq $10, -8(%rbp)` stores 10 in variable 1. The instruction `negq -8(%rbp)` changes variable 1 to -10. The instruction `movq $52, %rax` places 52 in the register `rax` and finally `addq -8(%rbp), %rax` adds the contents of variable 1 to `rax`, at which point `rax` contains 42.

The three instructions under the label `conclusion` are the typical *conclusion* of a procedure. The first two instructions are necessary to get the state of the machine back to where it was at the beginning of the procedure. The instruction `addq $16, %rsp` moves the stack pointer back to point at the old base pointer. The amount added here needs to match the amount that was subtracted in the prelude of the procedure. Then `popq %rbp` returns the old base pointer to `rbp` and adds 8 to the stack pointer. The last instruction, `retq`, jumps back to the procedure that called this one and adds 8 to the stack pointer, which returns the stack pointer to where it was prior to the procedure call.

The compiler needs a convenient representation for manipulating x86 programs, so we define an abstract syntax for x86 in Figure 2.8. We refer to this language as x86_0 with a subscript 0 because later we introduce extended versions of this assembly language. The main difference compared to the concrete syntax of x86 (Figure 2.4) is that it does not allow labeled instructions to appear anywhere, but instead organizes instructions into a group called a *block* and associates a label with every block, which is why

```
reg     ::= rsp | rbp | rax | rbx | rcx | rdx | rsi | rdi |
            r8 | r9 | r10 | r11 | r12 | r13 | r14 | r15
arg     ::= (Imm int) | (Reg reg) | (Deref reg int)
instr   ::= (Instr 'addq (list arg arg)) | (Instr 'subq (list arg arg))
          | (Instr 'movq (list arg arg)) | (Instr 'negq (list arg))
          | (Callq label) | (Retq) | (Pushq arg) | (Popq arg) | (Jmp label)
block   ::= (Block info instr ...)
x86_0   ::= (Program info (CFG (label . block) ...))
```

Figure 2.8: The abstract syntax of x86_0 assembly.

the `CFG` struct (for control-flow graph) includes an alist mapping labels to blocks. The reason for this organization becomes apparent in Chapter 4 when we introduce conditional branching. The `Block` structure includes an *info* field that is not needed for this chapter, but will become useful in Chapter 3. For now, the *info* field should just contain an empty list.

2.3 Planning the trip to x86 via the C_0 language

To compile one language to another it helps to focus on the differences between the two languages because the compiler will need to bridge those differences. What are the differences between R_1 and x86 assembly? Here are some of the most important ones:

(a) x86 arithmetic instructions typically have two arguments and update the second argument in place. In contrast, R_1 arithmetic operations take two arguments and produce a new value. An x86 instruction may have at most one memory-accessing argument. Furthermore, some instructions place special restrictions on their arguments.

(b) An argument of an R_1 operator can be any expression, whereas x86 instructions restrict their arguments to be integers constants, registers, and memory locations.

(c) The order of execution in x86 is explicit in the syntax: a sequence of instructions and jumps to labeled positions, whereas in R_1 the order of evaluation is a left-to-right depth-first traversal of the abstract syntax tree.

(d) An R_1 program can have any number of variables whereas x86 has 16 registers and the procedure calls stack.

(e) Variables in R_1 can overshadow other variables with the same name. The registers and memory locations of x86 all have unique names or addresses.

We ease the challenge of compiling from R_1 to x86 by breaking down the problem into several steps, dealing with the above differences one at a time. Each of these steps is called a *pass* of the compiler. This terminology comes from each step traverses (i.e. passes over) the AST of the program. We begin by sketching how we might implement each pass, and give them names. We then figure out an ordering of the passes and the input/output language for each pass. The very first pass has R_1 as its input language and the last pass has x86 as its output language. In between we can choose whichever language is most convenient for expressing the output of each pass, whether that be R_1, x86, or new *intermediate languages* of our own design. Finally, to implement each pass we write one recursive function per non-terminal in the grammar of the input language of the pass.

Pass `select-instructions` To handle the difference between R_1 operations and x86 instructions we convert each R_1 operation to a short sequence of instructions that accomplishes the same task.

Pass `remove-complex-opera*` To ensure that each subexpression (i.e. operator and operand, and hence the name `opera*`) is an *atomic* expression (a variable or integer), we introduce temporary variables to hold the results of subexpressions.

Pass `explicate-control` To make the execution order of the program explicit, we convert from the abstract syntax tree representation into a control-flow graph in which each node contains a sequence of statements and the edges between nodes say where to go at the end of the sequence.

Pass `assign-homes` To handle the difference between the variables in R_1 versus the registers and stack locations in x86, we map each variable to a register or stack location.

Pass `uniquify` This pass deals with the shadowing of variables by renaming every variable to a unique name, so that shadowing no longer occurs.

The next question is: in what order should we apply these passes? This question can be challenging because it is difficult to know ahead of time which orders will be better (easier to implement, produce more efficient

code, etc.) so oftentimes trial-and-error is involved. Nevertheless, we can try to plan ahead and make educated choices regarding the ordering.

Let us consider the ordering of `uniquify` and `remove-complex-opera*`. The assignment of subexpressions to temporary variables involves introducing new variables and moving subexpressions, which might change the shadowing of variables and inadvertently change the behavior of the program. But if we apply `uniquify` first, this will not be an issue. Of course, this means that in `remove-complex-opera*`, we need to ensure that the temporary variables that it creates are unique.

What should be the ordering of `explicate-control` with respect to `uniquify`? The `uniquify` pass should come first because `explicate-control` changes all the `let`-bound variables to become local variables whose scope is the entire program, which would confuse variables with the same name. Likewise, we place `explicate-control` after `remove-complex-opera*` because `explicate-control` removes the `let` form, but it is convenient to use `let` in the output of `remove-complex-opera*`. Regarding `assign-homes`, it is helpful to place `explicate-control` first because `explicate-control` changes `let`-bound variables into program-scope variables. This means that the `assign-homes` pass can read off the variables from the *info* of the `Program` AST node instead of traversing the entire program in search of `let`-bound variables.

Last, we need to decide on the ordering of `select-instructions` and `assign-homes`. These two passes are intertwined, creating a Gordian Knot. To do a good job of assigning homes, it is helpful to have already determined which instructions will be used, because x86 instructions have restrictions about which of their arguments can be registers versus stack locations. One might want to give preferential treatment to variables that occur in register-argument positions. On the other hand, it may turn out to be impossible to make sure that all such variables are assigned to registers, and then one must redo the selection of instructions. Some compilers handle this problem by iteratively repeating these two passes until a good solution is found. We shall use a simpler approach in which `select-instructions` comes first, followed by the `assign-homes`, then a third pass named `patch-instructions` that uses a reserved register to patch-up outstanding problems regarding instructions with too many memory accesses. The disadvantage of this approach is some programs may not execute as efficiently as they would if we used the iterative approach and used all of the registers for variables.

Figure 2.9 presents the ordering of the compiler passes in the form of a graph. Each pass is an edge and the input/output language of each pass is a node in the graph. The output of `uniquify` and `remove-complex-opera*`

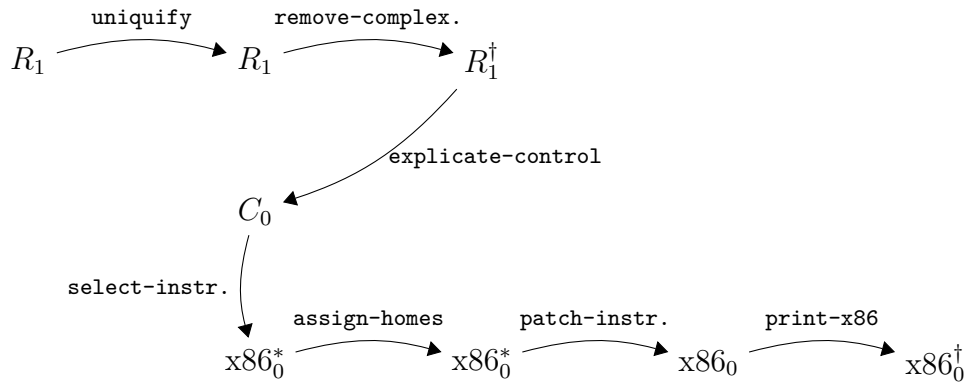


Figure 2.9: Overview of the passes for compiling R_1.

are programs that are still in the R_1 language, but the output of the pass `explicate-control` is in a different language C_0 that is designed to make the order of evaluation explicit in its syntax, which we introduce in the next section. The `select-instruction` pass translates from C_0 to a variant of x86. The `assign-homes` and `patch-instructions` passes input and output variants of x86 assembly. The last pass in Figure 2.9 is `print-x86`, which converts from the abstract syntax of x86_0 to the concrete syntax of x86.

In the next sections we discuss the C_0 language and the x86_0^* and $\text{x86}_0^\dagger$ dialects of x86. The remainder of this chapter gives hints regarding the implementation of each of the compiler passes in Figure 2.9.

2.3.1 The C_0 Intermediate Language

The output of `explicate-control` is similar to the *C* language [Kernighan and Ritchie, 1988] in that it has separate syntactic categories for expressions and statements, so we name it C_0. The concrete syntax for C_0 is defined in Figure 2.10 and the abstract syntax for C_0 is defined in Figure 2.11. The C_0 language supports the same operators as R_1 but the arguments of operators are restricted to atomic expressions (variables and integers), thanks to the `remove-complex-opera*` pass. Instead of `Let` expressions, C_0 has assignment statements which can be executed in sequence using the `Seq` form. A sequence of statements always ends with `Return`, a guarantee that is baked into the grammar rules for the *tail* non-terminal. The naming of this non-terminal comes from the term *tail position*, which refers to an expression that is the last one to execute within a function. (A expression in tail position may contain subexpressions, and those may or may not be

atm	::=	*int* | *var*
exp	::=	*atm* | `(read)` | `(-` *atm*`)` | `(+` *atm atm*`)`
stmt	::=	*var* `=` *exp*`;`
tail	::=	`return` *exp*`;` | *stmt tail*
C_0	::=	(*label*`:` *tail*) ...

Figure 2.10: The concrete syntax of the C_0 intermediate language.

atm	::=	`(Int` *int*`)` | `(Var` *var*`)`
exp	::=	*atm* | `(Prim 'read '())` | `(Prim '-` `(list` *atm* `))`
	|	`(Prim '+` `(list` *atm atm*`))`
stmt	::=	`(Assign (Var` *var*`)` *exp*`)`
tail	::=	`(Return` *exp*`)` | `(Seq` *stmt tail*`)`
C_0	::=	`(Program` *info* `(CFG (`*label* `.` *tail*`) ...))`

Figure 2.11: The abstract syntax of the C_0 intermediate language.

in tail position depending on the kind of expression.)

A C_0 program consists of a control-flow graph (represented as an alist mapping labels to tails). This is more general than necessary for the present chapter, as we do not yet need to introduce `goto` for jumping to labels, but it saves us from having to change the syntax of the program construct in Chapter 4. For now there will be just one label, `start`, and the whole program is its tail. The *info* field of the `Program` form, after the `explicate-control` pass, contains a mapping from the symbol `locals` to a list of variables, that is, a list of all the variables used in the program. At the start of the program, these variables are uninitialized; they become initialized on their first assignment.

2.3.2 The dialects of x86

The x86_0^* language, pronounced "pseudo x86", is the output of the pass `select-instructions`. It extends x86_0 with an unbounded number of program-scope variables and has looser rules regarding instruction arguments. The $\text{x86}^\dagger$ language, the output of `print-x86`, is the concrete syntax for x86.

2.4 Uniquify Variables

The `uniquify` pass compiles arbitrary R_1 programs into R_1 programs in which every `let` uses a unique variable name. For example, the `uniquify` pass should translate the program on the left into the program on the right.

```
(let ([x 32])                    ⇒   (let ([x.1 32])
  (+ (let ([x 10]) x) x))              (+ (let ([x.2 10]) x.2) x.1))
```

The following is another example translation, this time of a program with a `let` nested inside the initializing expression of another `let`.

```
(let ([x (let ([x 4])              (let ([x.2 (let ([x.1 4])
           (+ x 1))])         ⇒               (+ x.1 1))])
  (+ x 2))                           (+ x.2 2))
```

We recommend implementing `uniquify` by creating a function named `uniquify-exp` that is structurally recursive function and mostly just copies the input program. However, when encountering a `let`, it should generate a unique name for the variable (the Racket function `gensym` is handy for this) and associate the old name with the new unique name in an alist. The `uniquify-exp` function will need to access this alist when it gets to a variable reference, so we add another parameter to `uniquify-exp` for the alist.

The skeleton of the `uniquify-exp` function is shown in Figure 2.12. The function is curried so that it is convenient to partially apply it to a symbol table and then apply it to different expressions, as in the last clause for primitive operations in Figure 2.12. The `for/list` form is useful for applying a function to each element of a list to produce a new list.

Exercise 1. Complete the `uniquify` pass by filling in the blanks, that is, implement the clauses for variables and for the `let` form.

Exercise 2. Test your `uniquify` pass by creating five example R_1 programs and checking whether the output programs produce the same result as the input programs. The R_1 programs should be designed to test the most interesting parts of the `uniquify` pass, that is, the programs should include `let` forms, variables, and variables that overshadow each other. The five programs should be in a subdirectory named `tests` and they should have the same file name except for a different integer at the end of the name, followed by the ending `.rkt`. Use the `interp-tests` function (Appendix 12.2) from `utilities.rkt` to test your `uniquify` pass on the example programs. See the `run-tests.rkt` script in the student support code for an example of how to use `interp-tests`.

```
(define (uniquify-exp symtab)
  (lambda (e)
    (match e
      [(Var x) ___]
      [(Int n) (Int n)]
      [(Let x e body) ___]
      [(Prim op es)
       (Prim op (for/list ([e es]) ((uniquify-exp symtab) e)))]
      )))

(define (uniquify p)
  (match p
    [(Program '() e)
     (Program '() ((uniquify-exp '()) e))]
    )))
```

Figure 2.12: Skeleton for the `uniquify` pass.

atm	::=	`(Int` *int*`)` | `(Var` *var*`)`
exp	::=	*atm* | `(Prim 'read '())`
	|	`(Prim '- (list` *atm* `))` | `(Prim '+ (list` *atm atm*`))`
	|	`(Let` *var exp exp*`)`
$R_1^\dagger$	::=	`(Program '()` *exp*`)`

Figure 2.13: $R_1^\dagger$ is R_1 in administrative normal form (ANF).

2.5 Remove Complex Operands

The `remove-complex-opera*` pass compiles R_1 programs into R_1 programs in which the arguments of operations are atomic expressions. Put another way, this pass removes complex operands, such as the expression `(- 10)` in the program below. This is accomplished by introducing a new `let`-bound variable, binding the complex operand to the new variable, and then using the new variable in place of the complex operand, as shown in the output of `remove-complex-opera*` on the right.

```
(+ 52 (- 10))
```

$\Rightarrow$

```
(let ([tmp.1 (- 10)])
  (+ 52 tmp.1))
```

Figure 2.13 presents the grammar for the output of this pass, language $R_1^\dagger$. The main difference is that operator arguments are required to be

atomic expressions. In the literature this is called *administrative normal form*, or ANF for short [Danvy, 1991, Flanagan et al., 1993].

We recommend implementing this pass with two mutually recursive functions, `rco-atom` and `rco-exp`. The idea is to apply `rco-atom` to subexpressions that are required to be atomic and to apply `rco-exp` to subexpressions that can be atomic or complex (see Figure 2.13). Both functions take an R_1 expression as input. The `rco-exp` function returns an expression. The `rco-atom` function returns two things: an atomic expression and alist mapping temporary variables to complex subexpressions. You can return multiple things from a function using Racket's `values` form and you can receive multiple things from a function call using the `define-values` form. If you are not familiar with these features, review the Racket documentation. Also, the `for/lists` form is useful for applying a function to each element of a list, in the case where the function returns multiple values.

The following shows the output of `rco-atom` on the expression `(- 10)` (using concrete syntax to be concise).

```
(- 10)                    ⇒   tmp.1
                              ((tmp.1 . (- 10)))
```

Take special care of programs such as the next one that `let`-bind variables with integers or other variables. You should leave them unchanged, as shown in to the program on the right

```
(let ([a 42])                    (let ([a 42])
  (let ([b a])            ⇒        (let ([b a])
    b))                              b))
```

A careless implementation of `rco-exp` and `rco-atom` might produce the following output.

```
(let ([tmp.1 42])
  (let ([a tmp.1])
    (let ([tmp.2 a])
      (let ([b tmp.2])
        b))))
```

Exercise 3. Implement the `remove-complex-opera*` pass. Test the new pass on all of the example programs that you created to test the `uniquify` pass and create three new example programs that are designed to exercise the interesting code in the `remove-complex-opera*` pass. Use the `interp-tests` function (Appendix 12.2) from `utilities.rkt` to test your passes on the example programs.

2.6 Explicate Control

The `explicate-control` pass compiles R_1 programs into C_0 programs that make the order of execution explicit in their syntax. For now this amounts to flattening `let` constructs into a sequence of assignment statements. For example, consider the following R_1 program.

```
(let ([y (let ([x 20])
           (+ x (let ([x 22]) x)))])
  y)
```

The output of the previous pass and of `explicate-control` is shown below. Recall that the right-hand-side of a `let` executes before its body, so the order of evaluation for this program is to assign `20` to `x.1`, assign `22` to `x.2`, assign `(+ x.1 x.2)` to `y`, then return `y`. Indeed, the output of `explicate-control` makes this ordering explicit.

```
(let ([y (let ([x.1 20])
           (let ([x.2 22])
             (+ x.1 x.2)))])
  y)
```

$\Rightarrow$

```
locals: y x.1 x.2
start:
  x.1 = 20;
  x.2 = 22;
  y = (+ x.1 x.2);
  return y;
```

We recommend implementing `explicate-control` using two mutually recursive functions: `explicate-tail` and `explicate-assign`. The first function should be applied to expressions in tail position whereas the second should be applied to expressions that occur on the right-hand-side of a `let`. The `explicate-tail` function takes an R_1 expression as input and produces a C_0 *tail* (see Figure 2.11) and a list of formerly `let`-bound variables. The `explicate-assign` function takes an R_1 expression, the variable that it is to be assigned to, and C_0 code (a *tail*) that should come after the assignment (e.g., the code generated for the body of the `let`). It returns a *tail* and a list of variables. The `explicate-assign` function is in accumulator-passing style in that its third parameter is some C_0 code which it then adds to and returns. The reader might be tempted to instead organize `explicate-assign` in a more direct fashion, without the third parameter and perhaps using `append` to combine statements. We warn against that alternative because the accumulator-passing style is key to how we generate high-quality code for conditional expressions in Chapter 4.

The top-level `explicate-control` function should invoke `explicate-tail` on the body of the `program` and then associate the `locals` symbol with the resulting list of variables in the *info* field, as in the above example.

2.7 Select Instructions

In the `select-instructions` pass we begin the work of translating from C_0 to x86_0^*. The target language of this pass is a variant of x86 that still uses variables, so we add an AST node of the form `(Var` *var*`)` to the x86_0 abstract syntax of Figure 2.8. We recommend implementing the `select-instructions` in terms of three auxiliary functions, one for each of the non-terminals of C_0: *atm*, *stmt*, and *tail*.

The cases for *atm* are straightforward, variables stay the same and integer constants are changed to immediates: `(Int` n`)` changes to `(Imm` n`)`.

Next we consider the cases for *stmt*, starting with arithmetic operations. For example, in C_0 an addition operation can take the form below, to the left of the $\Rightarrow$. To translate to x86, we need to use the `addq` instruction which does an in-place update. So we must first move `10` to `x`.

```
x = (+ 10 32);          ⇒   movq $10, x
                            addq $32, x
```

There are cases that require special care to avoid generating needlessly complicated code. If one of the arguments of the addition is the same as the left-hand side of the assignment, then there is no need for the extra move instruction. For example, the following assignment statement can be translated into a single `addq` instruction.

```
x = (+ 10 x);           ⇒   addq $10, x
```

The `read` operation does not have a direct counterpart in x86 assembly, so we have instead implemented this functionality in the C language [Kernighan and Ritchie, 1988], with the function `read_int` in the file `runtime.c`. In general, we refer to all of the functionality in this file as the *runtime system*, or simply the *runtime* for short. When compiling your generated x86 assembly code, you need to compile `runtime.c` to `runtime.o` (an "object file", using `gcc` option `-c`) and link it into the executable. For our purposes of code generation, all you need to do is translate an assignment of `read` into some variable *lhs* (for left-hand side) into a call to the `read_int` function followed by a move from `rax` to the left-hand side. The move from `rax` is needed because the return value from `read_int` goes into `rax`, as is the case in general.

var `= (read);` ⇒

```
callq read_int
movq %rax, var
```

There are two cases for the *tail* non-terminal: `Return` and `Seq`. Regard-

ing `Return`, we recommend treating it as an assignment to the `rax` register followed by a jump to the conclusion of the program (so the conclusion needs to be labeled). For `(Seq` s t`)`, you can translate the statement s and tail t recursively and append the resulting instructions.

Exercise 4. Implement the `select-instructions` pass and test it on all of the example programs that you created for the previous passes and create three new example programs that are designed to exercise all of the interesting code in this pass. Use the `interp-tests` function (Appendix 12.2) from `utilities.rkt` to test your passes on the example programs.

2.8 Assign Homes

The `assign-homes` pass compiles x86_0^* programs to x86_0^* programs that no longer use program variables. Thus, the `assign-homes` pass is responsible for placing all of the program variables in registers or on the stack. For runtime efficiency, it is better to place variables in registers, but as there are only 16 registers, some programs must necessarily resort to placing some variables on the stack. In this chapter we focus on the mechanics of placing variables on the stack. We study an algorithm for placing variables in registers in Chapter 3.

Consider again the following R_1 program.

```
(let ([a 42])
  (let ([b a])
    b))
```

For reference, we repeat the output of `select-instructions` on the left and show the output of `assign-homes` on the right. Recall that `explicate-control` associated the list of variables with the `locals` symbol in the program's *info* field, so `assign-homes` has convenient access to the them. In this example, we assign variable `a` to stack location `-8(%rbp)` and variable `b` to location `-16(%rbp)`.

```
locals: a b
start:
    movq $42, a
    movq a, b
    movq b, %rax
    jmp conclusion
```

$\Rightarrow$

```
stack-space: 16
start:
    movq $42, -8(%rbp)
    movq -8(%rbp), -16(%rbp)
    movq -16(%rbp), %rax
    jmp conclusion
```

In the process of assigning variables to stack locations, it is convenient to compute and store the size of the frame (in bytes) in the *info* field of

the `Program` node, with the key `stack-space`, which will be needed later to generate the procedure conclusion. The x86-64 standard requires the frame size to be a multiple of 16 bytes.

Exercise 5. Implement the `assign-homes` pass and test it on all of the example programs that you created for the previous passes pass. We recommend that `assign-homes` take an extra parameter that is a mapping of variable names to homes (stack locations for now). Use the `interp-tests` function (Appendix 12.2) from `utilities.rkt` to test your passes on the example programs.

2.9 Patch Instructions

The `patch-instructions` pass compiles x86_0^* programs to x86_0 programs by making sure that each instruction adheres to the restrictions of the x86 assembly language. In particular, at most one argument of an instruction may be a memory reference.

We return to the following running example.

```
(let ([a 42])
  (let ([b a])
    b))
```

After the `assign-homes` pass, the above program has been translated to the following.

```
stack-space: 16
start:
    movq $42, -8(%rbp)
    movq -8(%rbp), -16(%rbp)
    movq -16(%rbp), %rax
    jmp conclusion
```

The second `movq` instruction is problematic because both arguments are stack locations. We suggest fixing this problem by moving from the source location to the register `rax` and then from `rax` to the destination location, as follows.

```
movq -8(%rbp), %rax
movq %rax, -16(%rbp)
```

Exercise 6. Implement the `patch-instructions` pass and test it on all of the example programs that you created for the previous passes and create

three new example programs that are designed to exercise all of the interesting code in this pass. Use the `interp-tests` function (Appendix 12.2) from `utilities.rkt` to test your passes on the example programs.

2.10 Print x86

The last step of the compiler from R_1 to x86 is to convert the x86_0 AST (defined in Figure 2.8) to the string representation (defined in Figure 2.4). The Racket `format` and `string-append` functions are useful in this regard. The main work that this step needs to perform is to create the `main` function and the standard instructions for its prelude and conclusion, as shown in Figure 2.6 of Section 2.2. You need to know the number of stack-allocated variables, so we suggest computing it in the `assign-homes` pass (Section 2.8) and storing it in the *info* field of the `program` node.

If you want your program to run on Mac OS X, your code needs to determine whether or not it is running on a Mac, and prefix underscores to labels like `main`. You can determine the platform with the Racket call `(system-type 'os)`, which returns `'macosx`, `'unix`, or `'windows`.

Exercise 7. Implement the `print-x86` pass and test it on all of the example programs that you created for the previous passes. Use the `compiler-tests` function (Appendix 12.2) from `utilities.rkt` to test your complete compiler on the example programs. See the `run-tests.rkt` script in the student support code for an example of how to use `compiler-tests`. Also, remember to compile the provided `runtime.c` file to `runtime.o` using `gcc`.

2.11 Challenge: Partial Evaluator for R_1

This section describes optional challenge exercises that involve adapting and improving the partial evaluator for R_0 that was introduced in Section 1.6.

Exercise 8. Adapt the partial evaluator from Section 1.6 (Figure 1.4) so that it applies to R_1 programs instead of R_0 programs. Recall that R_1 adds `let` binding and variables to the R_0 language, so you will need to add cases for them in the `pe-exp` function. Also, note that the `program` form changes slightly to include an *info* field. Once complete, add the partial evaluation pass to the front of your compiler and make sure that your compiler still passes all of the tests.

The next exercise builds on Exercise 8.

Exercise 9. Improve on the partial evaluator by replacing the `pe-neg` and `pe-add` auxiliary functions with functions that know more about arithmetic. For example, your partial evaluator should translate

```
(+ 1 (+ (read) 1))
```

into

```
(+ 2 (read))
```

To accomplish this, the `pe-exp` function should produce output in the form of the *residual* non-terminal of the following grammar.

$$\begin{array}{lcl} \textit{inert} & ::= & \textit{var} \mid (\texttt{read}) \mid (\texttt{-}\ (\texttt{read})) \mid (\texttt{+}\ \textit{inert}\ \textit{inert}) \\ \textit{residual} & ::= & \textit{int} \mid (\texttt{+}\ \textit{int}\ \textit{inert}) \mid \textit{inert} \end{array}$$

The `pe-add` and `pe-neg` functions may therefore assume that their inputs are *residual* expressions and they should return *residual* expressions. Once the improvements are complete, make sure that your compiler still passes all of the tests. After all, fast code is useless if it produces incorrect results!

3

Register Allocation

In Chapter 2 we placed all variables on the stack to make our life easier. However, we can improve the performance of the generated code if we instead place some variables into registers. The CPU can access a register in a single cycle, whereas accessing the stack takes many cycles if the relevant data is in cache or many more to access main memory if the data is not in cache. Figure 3.1 shows a program with four variables that serves as a running example. We show the source program and also the output of instruction selection. At that point the program is almost x86 assembly but not quite; it still contains variables instead of stack locations or registers.

The goal of register allocation is to fit as many variables into registers as possible. A program sometimes has more variables than registers, so we cannot map each variable to a different register. Fortunately, it is common for different variables to be needed during different periods of time during program execution, and in such cases several variables can be mapped to the same register. Consider variables `x` and `y` in Figure 3.1. After the variable `x` is moved to `z` it is no longer needed. Variable `y`, on the other hand, is used only after this point, so `x` and `y` could share the same register. The topic of Section 3.2 is how to compute where a variable is needed. Once we have that information, we compute which variables are needed at the same time, i.e., which ones *interfere* with each other, and represent this relation as an undirected graph whose vertices are variables and edges indicate when two variables interfere (Section 3.3). We then model register allocation as a graph coloring problem, which we discuss in Section 3.4.

In the event that we run out of registers despite these efforts, we place the remaining variables on the stack, similar to what we did in Chapter 2. It is common to use the verb *spill* for assigning a variable to a stack location.

Example R_1 program:

```
(let ([v 1])
  (let ([w 42])
    (let ([x (+ v 7)])
      (let ([y x])
        (let ([z (+ x w)])
          (+ z (- y)))))))
```

After instruction selection:

```
locals: (v w x y z t)
start:
    movq $1, v
    movq $42, w
    movq v, x
    addq $7, x
    movq x, y
    movq x, z
    addq w, z
    movq y, t
    negq t
    movq z, %rax
    addq t, %rax
    jmp conclusion
```

Figure 3.1: A running example program for register allocation.

The process of spilling variables is handled as part of the graph coloring process described in 3.4.

We make the simplifying assumption that each variable is assigned to one location (a register or stack address). A more sophisticated approach is to assign a variable to one or more locations in different regions of the program. For example, if a variable is used many times in short sequence and then only used again after many other instructions, it could be more efficient to assign the variable to a register during the intial sequence and then move it to the stack for the rest of its lifetime. We refer the interested reader to Cooper and Simpson [1998] and Cooper and Torczon [2011] for more information about this approach.

3.1 Registers and Calling Conventions

As we perform register allocation, we need to be aware of the conventions that govern the way in which registers interact with function calls, such as calls to the `read_int` function in our generated code and even the call that the operating system makes to execute our `main` function. The convention for x86 is that the caller is responsible for freeing up some registers, the *caller-saved registers*, prior to the function call, and the callee is responsible for preserving the values in some other registers, the *callee-saved registers*. The caller-saved registers are

```
rax rcx rdx rsi rdi r8 r9 r10 r11
```

while the callee-saved registers are

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

We can think about this caller/callee convention from two points of view, the caller view and the callee view:

- The caller should assume that all the caller-saved registers get overwritten with arbitrary values by the callee. On the other hand, the caller can safely assume that all the callee-saved registers contain the same values after the call that they did before the call.
- The callee can freely use any of the caller-saved registers. However, if the callee wants to use a callee-saved register, the callee must arrange to put the original value back in the register prior to returning to the caller, which is usually accomplished by saving the value to the stack in the prelude of the function and restoring the value in the conclusion of the function.

The next question is how these calling conventions impact register allocation. Consider the R_1 program in Figure 3.2. We first analyze this example from the caller point of view and then from the callee point of view.

The program makes two calls to the `read` function. Also, the variable `x` is in-use during the second call to `read`, so we need to make sure that the value in `x` does not get accidentally wiped out by the call to `read`. One obvious approach is to save all the values in caller-saved registers to the stack prior to each function call, and restore them after each call. That way, if the register allocator chooses to assign `x` to a caller-saved register, its value will be preserved accross the call to `read`. However, the disadvantage of this approach is that saving and restoring to the stack is relatively slow. If `x` is not used many times, it may be better to assign `x` to a stack location in the first place. Or better yet, if we can arrange for `x` to be placed in a callee-saved register, then it won't need to be saved and restored during function calls.

The approach that we recommend for variables that are in-use during a function call is to either assign them to callee-saved registers or to spill them to the stack. On the other hand, for variables that are not in-use during a function call, we try the following alternatives in order 1) look for an available caller-saved register (to leave room for other variables in the callee-saved register), 2) look for a callee-saved register, and 3) spill the variable to the stack.

It is straightforward to implement this approach in a graph coloring register allocator. First, we know which variables are in-use during every function call because we compute that information for every instruction (Section 3.2). Second, when we build the interference graph (Section 3.3), we can place an edge between each of these variables and the caller-saved registers in the interference graph. This will prevent the graph coloring algorithm from assigning those variables to caller-saved registers.

Returning to the example in Figure 3.2, let us analyze the generated x86 code on the right-hand side, focusing on the `start` block. Notice that variable `x` is assigned to `rbx`, a callee-saved register. Thus, it is already in a safe place during the second call to `read_int`. Next, notice that variable `y` is assigned to `rcx`, a caller-saved register, because there are no function calls in the remainder of the block.

Next we analyze the example from the callee point of view, focusing on the prelude and conclusion of the `main` function. As usual the prelude begins with saving the `rbp` register to the stack and setting the `rbp` to the current stack pointer. We now know why it is necessary to save the `rbp`: it is a callee-saved register. The prelude then pushes `rbx` to the stack because 1) `rbx` is also a callee-saved register and 2) `rbx` is assigned to a variable (`x`). There are several more callee-saved register that are not saved in the prelude because they were not assigned to variables. The prelude subtracts 8 bytes from the `rsp` to make it 16-byte aligned and then jumps to the `start` block. Shifting attention to the `conclusion`, we see that `rbx` is restored from the stack with a `popq` instruction.

Example R_1 program:

```
(let ([x (read)])
  (let ([y (read)])
    (+ (+ x y) 42)))
```

Generated x86 assembly:

```
start:
        callq  read_int
        movq   %rax, %rbx
        callq  read_int
        movq   %rax, %rcx
        addq   %rcx, %rbx
        movq   %rbx, %rax
        addq   $42, %rax
        jmp _conclusion

        .globl main
main:
        pushq  %rbp
        movq   %rsp, %rbp
        pushq  %rbx
        subq   $8, %rsp
        jmp start
conclusion:
        addq   $8, %rsp
        popq   %rbx
        popq   %rbp
        retq
```

Figure 3.2: An example with function calls.

3.2 Liveness Analysis

A variable or register is *live* at a program point if its current value is used at some later point in the program. We shall refer to variables and registers collectively as *locations*. Consider the following code fragment in which there are two writes to `b`. Are `a` and `b` both live at the same time?

```
1 movq $5, a
2 movq $30, b
3 movq a, c
4 movq $10, b
5 addq b, c
```

The answer is no because the integer `30` written to `b` on line 2 is never used. The variable `b` is read on line 5 and there is an intervening write to `b` on line 4, so the read on line 5 receives the value written on line 4, not line 2.

The live locations can be computed by traversing the instruction sequence back to front (i.e., backwards in execution order). Let $I_1, \ldots, I_n$ be the instruction sequence. We write $L_{\mathsf{after}}(k)$ for the set of live locations after instruction I_k and $L_{\mathsf{before}}(k)$ for the set of live locations before instruction I_k. The live locations after an instruction are always the same as the live locations before the next instruction.

$$L_{\mathsf{after}}(k) = L_{\mathsf{before}}(k+1) \tag{3.1}$$

To start things off, there are no live locations after the last instruction[1], so

$$L_{\mathsf{after}}(n) = \emptyset \tag{3.2}$$

We then apply the following rule repeatedly, traversing the instruction sequence back to front.

$$L_{\mathtt{before}}(k) = (L_{\mathtt{after}}(k) - W(k)) \cup R(k), \tag{3.3}$$

where $W(k)$ are the locations written to by instruction I_k and $R(k)$ are the locations read by instruction I_k.

[1] Technically, the `rax` register is live but we do not use it for register allocation.

The Racket Set Package

A *set* is an unordered collection of elements without duplicates.

- **`(set`** $v \ldots$**`)`** constructs a set containing the specified elements.
- **`(set-union`** set_1 set_2**`)`** returns the union of the two sets.
- **`(set-subtract`** set_1 set_2**`)`** returns the difference of the two sets.
- **`(set-member?`** set v**`)`** is element v in set?
- **`(set-count`** set**`)`** how many unique elements are in set?
- **`(set->list`** set**`)`** converts the set to a list.

Let us walk through the above example, applying these formulas starting with the instruction on line 5. We collect the answers in the below listing. The L_{after} for the `addq b, c` instruction is $\emptyset$ because it is the last instruction (formula 3.2). The L_{before} for this instruction is $\{\mathtt{b}, \mathtt{c}\}$ because it reads from variables `b` and `c` (formula 3.3), that is

$$L_{\mathsf{before}}(5) = (\emptyset - \{\mathtt{c}\}) \cup \{\mathtt{b}, \mathtt{c}\} = \{\mathtt{b}, \mathtt{c}\}$$

Moving on the the instruction `movq $10, b` at line 4, we copy the live-before set from line 5 to be the live-after set for this instruction (formula 3.1).

$$L_{\mathsf{after}}(4) = \{\mathtt{b}, \mathtt{c}\}$$

This move instruction writes to `b` and does not read from any variables, so we have the following live-before set (formula 3.3).

$$L_{\mathsf{before}}(4) = (\{\mathtt{b}, \mathtt{c}\} - \{\mathtt{b}\}) \cup \emptyset = \{\mathtt{c}\}$$

The live-before for instruction `movq a, c` is $\{\mathtt{a}\}$ because it writes to $\{\mathtt{c}\}$ and reads from $\{\mathtt{a}\}$ (formula 3.3). The live-before for `movq $30, b` is $\{\mathtt{a}\}$ because it writes to a variable that is not live and does not read from a variable. Finally, the live-before for `movq $5, a` is $\emptyset$ because it writes to variable `a`.

```
1 movq $5, a
2 movq $30, b
3 movq a, c
4 movq $10, b
5 addq b, c
```

$$\begin{aligned}
L_{\mathsf{before}}(1) = \emptyset, L_{\mathsf{after}}(1) &= \{\mathtt{a}\} \\
L_{\mathsf{before}}(2) = \{\mathtt{a}\}, L_{\mathsf{after}}(2) &= \{\mathtt{a}\} \\
L_{\mathsf{before}}(3) = \{\mathtt{a}\}, L_{\mathsf{after}}(2) &= \{\mathtt{c}\} \\
L_{\mathsf{before}}(4) = \{\mathtt{c}\}, L_{\mathsf{after}}(4) &= \{\mathtt{b}, \mathtt{c}\} \\
L_{\mathsf{before}}(5) = \{\mathtt{b}, \mathtt{c}\}, L_{\mathsf{after}}(5) &= \emptyset
\end{aligned}$$

Figure 3.3 shows the results of liveness analysis for the running example program, with the live-before and live-after sets shown between each instruction to make the figure easy to read.

Exercise 10. Implement the compiler pass named `uncover-live` that computes the live-after sets. We recommend storing the live-after sets (a list of a set of variables) in the *info* field of the `Block` structure. We recommend organizing your code to use a helper function that takes a list of instructions and an initial live-after set (typically empty) and returns the list of live-after sets. We recommend creating helper functions to 1) compute the set of locations that appear in an argument (of an instruction), 2) compute

```
                    {}
movq $1, v
                    {v}
movq $42, w
                    {v, w}
movq v, x
                    {w, x}
addq $7, x
                    {w, x}
movq x, y
                    {w, x, y}
movq x, z
                    {w, y, z}
addq w, z
                    {y, z}
movq y, t
                    {t, z}
negq t
                    {t, z}
movq z, %rax
                    {rax, t}
addq t, %rax
                    {}
jmp conclusion
                    {}
```

Figure 3.3: The running example annotated with live-after sets.

the locations read by an instruction which corresponds to the R function discussed above, and 3) the locations written by an instruction which corresponds to W. The `callq` instruction should include all of the caller-saved registers in its W because the calling convention says that those registers may be written to during the function call.

3.3 Building the Interference Graph

Based on the liveness analysis, we know where each variable is needed. However, during register allocation, we need to answer questions of the specific form: are variables u and v live at the same time? (And therefore cannot be assigned to the same register.) To make this question easier to answer, we create an explicit data structure, an *interference graph*. An interference graph is an undirected graph that has an edge between two variables if they are live at the same time, that is, if they interfere with each other.

The most obvious way to compute the interference graph is to look at the set of live location between each statement in the program and add an edge to the graph for every pair of variables in the same set. This approach is less than ideal for two reasons. First, it can be expensive because it takes $O(n^2)$ time to look at every pair in a set of n live locations. Second, there is a special case in which two locations that are live at the same time do not actually interfere with each other: when they both contain the same value because we have assigned one to the other.

A better way to compute the interference graph is to focus on the writes Appel and Palsberg [2003]. We do not want the writes performed by an instruction to overwrite something in a live location. So for each instruction, we create an edge between the locations being written to and all

The Racket Graph Library

A *graph* is a collection of vertices and edges where each edge connects two vertices. A graph is *directed* if each edge points from a source to a target. Otherwise the graph is *undirected*.

`(directed-graph` *edges*`)` constructs a directed graph from a list of edges. Each edge is a list containing the source and target vertex.

`(undirected-graph` *edges*`)` constructs a undirected graph from a list of edges. Each edge is represented by a list containing two vertices.

`(add-vertex!` *graph vertex*`)` inserts a vertex into the graph.

`(add-edge!` *graph source target*`)` inserts an edge between the two vertices into the graph.

`(in-neighbors` *graph vertex*`)` returns a sequence of all the neighbors of the given vertex.

`(in-vertices` *graph*`)` returns a sequence of all the vertices in the graph.

`movq $1, v`	no interference by rule 1
`movq $42, w`	`w` interferes with `v` by rule 1
`movq v, x`	`x` interferes with `w` by rule 1
`addq $7, x`	`x` interferes with `w` by rule 2
`movq x, y`	`y` interferes with `w` but not `x` by rule 1
`movq x, z`	`z` interferes with `w` and `y` by rule 1
`addq w, z`	`z` interferes with `y` by rule 2
`movq y, t`	`t` interferes with `z` by rule 1
`negq t`	`t` interferes with `z` by rule 2
`movq z, %rax`	`rax` interferes with `t` by rule 1
`addq t, %rax`	no interference by rule 2
`jmp conclusion`	no interference by rule 2

Figure 3.4: Interference results for the running example.

the other live locations. (Except that one should not create self edges.) Recall that for a `callq` instruction, we consider all of the caller-saved registers as being written to, so an edge will be added between every live variable and every caller-saved register. For `movq`, we deal with the above-mentioned special case by not adding an edge between a live variable v and destination d if v matches the source of the move. So we have the following two rules.

1. If instruction I_k is a move such as `movq` s`,` d, then add the edge (d, v) for every $v \in L_{\mathsf{after}}(k)$ unless $v = d$ or $v = s$.

2. For any other instruction I_k, for every $d \in W(k)$ add an edge (d, v) for every $v \in L_{\mathsf{after}}(k)$ unless $v = d$.

Working from the top to bottom of Figure 3.3, we apply the above rules to each instruction. We highlight a few of the instructions and then refer the reader to Figure 3.4 for all the interference results. The first instruction is `movq $1, v`, so rule 3 applies, and the live-after set is $\{\mathtt{v}\}$. We do not add any interference edges because the one live variable `v` is also the destination of this instruction. For the second instruction, `movq $42, w`, so rule 3 applies again, and the live-after set is $\{\mathtt{v}, \mathtt{w}\}$. So the target `w` of `movq` interferes with `v`. Next we skip forward to the instruction `movq x, y`.

The resulting interference graph is shown in Figure 3.5.

Exercise 11. Implement the compiler pass named `build-interference` according to the algorithm suggested above. We recommend using the `graph`

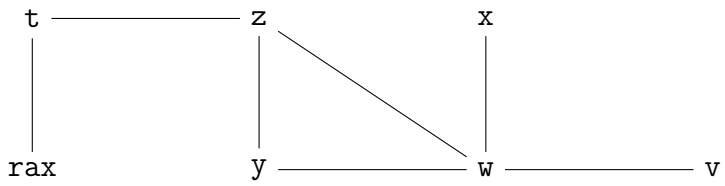


Figure 3.5: The interference graph of the example program.

package to create and inspect the interference graph. The output graph of this pass should be stored in the *info* field of the program, under the key `conflicts`.

3.4 Graph Coloring via Sudoku

We come to the main event, mapping variables to registers (or to stack locations in the event that we run out of registers). We need to make sure that two variables do not get mapped to the same register if the two variables interfere with each other. Thinking about the interference graph, this means that adjacent vertices must be mapped to different registers. If we think of registers as colors, the register allocation problem becomes the widely-studied graph coloring problem [Balakrishnan, 1996, Rosen, 2002].

The reader may be more familiar with the graph coloring problem than he or she realizes; the popular game of Sudoku is an instance of the graph coloring problem. The following describes how to build a graph out of an initial Sudoku board.

- There is one vertex in the graph for each Sudoku square.
- There is an edge between two vertices if the corresponding squares are in the same row, in the same column, or if the squares are in the same 3×3 region.
- Choose nine colors to correspond to the numbers 1 to 9.
- Based on the initial assignment of numbers to squares in the Sudoku board, assign the corresponding colors to the corresponding vertices in the graph.

If you can color the remaining vertices in the graph with the nine colors, then you have also solved the corresponding game of Sudoku. Figure 3.6 shows

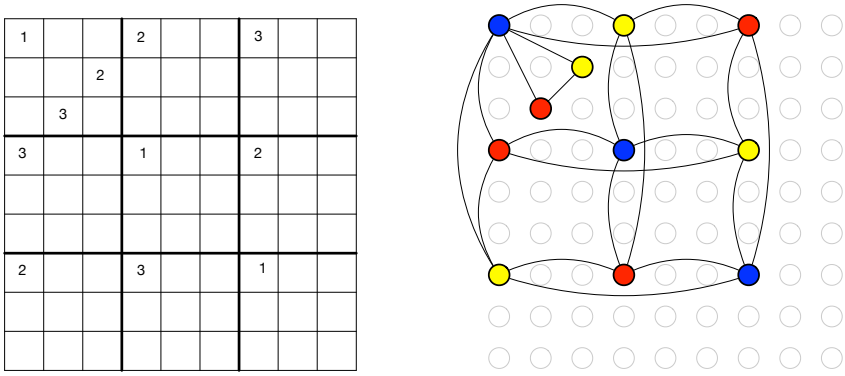


Figure 3.6: A Sudoku game board and the corresponding colored graph.

an initial Sudoku game board and the corresponding graph with colored vertices. We map the Sudoku number 1 to blue, 2 to yellow, and 3 to red. We only show edges for a sampling of the vertices (the colored ones) because showing edges for all of the vertices would make the graph unreadable.

Given that Sudoku is an instance of graph coloring, one can use Sudoku strategies to come up with an algorithm for allocating registers. For example, one of the basic techniques for Sudoku is called Pencil Marks. The idea is to use a process of elimination to determine what numbers no longer make sense for a square and write down those numbers in the square (writing very small). For example, if the number 1 is assigned to a square, then by process of elimination, you can write the pencil mark 1 in all the squares in the same row, column, and region. Many Sudoku computer games provide automatic support for Pencil Marks. The Pencil Marks technique corresponds to the notion of *saturation* due to Brélaz [1979]. The saturation of a vertex, in Sudoku terms, is the set of numbers that are no longer available. In graph terminology, we have the following definition:

$$\text{saturation}(u) = \{c \mid \exists v.v \in \text{neighbors}(u) \text{ and } \text{color}(v) = c\}$$

where neighbors(u) is the set of vertices that share an edge with u.

Using the Pencil Marks technique leads to a simple strategy for filling in numbers: if there is a square with only one possible number left, then choose that number! But what if there are no squares with only one possibility left? One brute-force approach is to try them all: choose the first and if it ultimately leads to a solution, great. If not, backtrack and choose the

Algorithm: DSATUR
Input: a graph G
Output: an assignment color[v] for each vertex $v \in G$

$W \leftarrow$ vertices(G)
while $W \neq \emptyset$ **do**
    pick a vertex u from W with the highest saturation,
        breaking ties randomly
    find the lowest color c that is not in $\{\text{color}[v] \ : \ v \in \text{adjacent}(u)\}$
    color[u] $\leftarrow c$
    $W \leftarrow W - \{u\}$

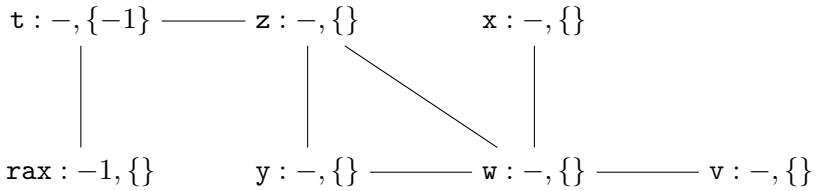
Figure 3.7: The saturation-based greedy graph coloring algorithm.

next possibility. One good thing about Pencil Marks is that it reduces the degree of branching in the search tree. Nevertheless, backtracking can be horribly time consuming. One way to reduce the amount of backtracking is to use the most-constrained-first heuristic. That is, when choosing a square, always choose one with the fewest possibilities left (the vertex with the highest saturation). The idea is that choosing highly constrained squares earlier rather than later is better because later on there may not be any possibilities left for those squares.

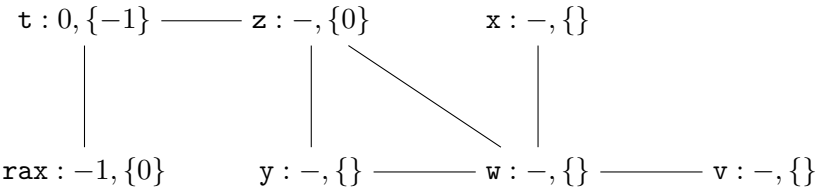
However, register allocation is easier than Sudoku because the register allocator can map variables to stack locations when the registers run out. Thus, it makes sense to drop backtracking in favor of greedy search, that is, make the best choice at the time and keep going. We still wish to minimize the number of colors needed, so keeping the most-constrained-first heuristic is a good idea. Figure 3.7 gives the pseudo-code for a simple greedy algorithm for register allocation based on saturation and the most-constrained-first heuristic. It is roughly equivalent to the DSATUR algorithm of Brélaz [1979] (also known as saturation degree ordering [Gebremedhin, 1999, Al-Omari and Sabri, 2006]). Just as in Sudoku, the algorithm represents colors with integers. The integers 0 through $k-1$ correspond to the k registers that we use for register allocation. The integers k and larger correspond to stack locations. The registers that are not used for register allocation, such as `rax`, are assigned to negative integers. In particular, we assign -1 to `rax`.

With this algorithm in hand, let us return to the running example and consider how to color the interference graph in Figure 3.5. We color the vertices for registers with their own color. For example, `rax` is assigned the

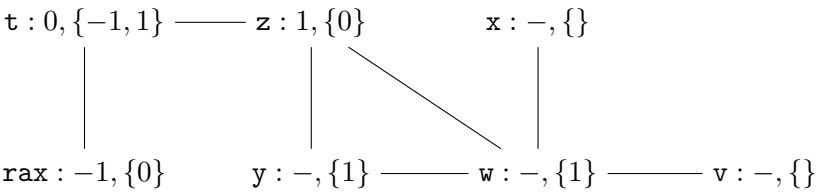
color -1. We then update the saturation for their neighboring vertices. In this case, the saturation for `t` includes -1. The remaining vertices are not yet colored, so they annotated with a dash, and their saturation sets are empty.



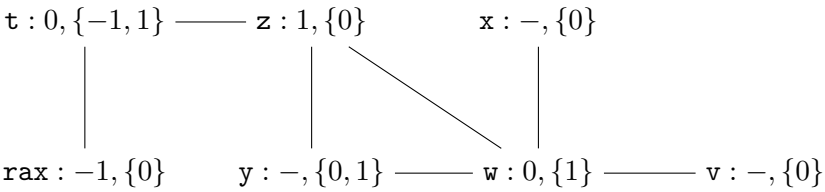
The algorithm says to select a maximally saturated vertex. So we pick `t` and color it with the first available integer, which is 0. We mark 0 as no longer available for `z` and `rax` because they interfere with `t`.



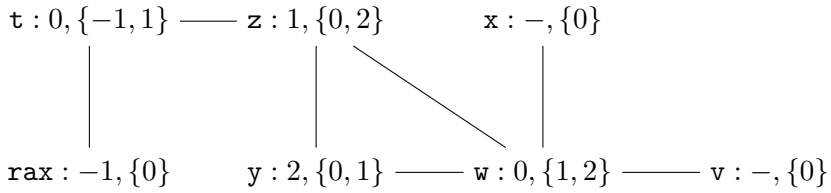
We repeat the process, selecting another maximally saturated vertex, which is `z`, and color it with the first available number, which is 1. We add 1 to the saturations for the neighboring vertices `t`, `y`, and `w`.



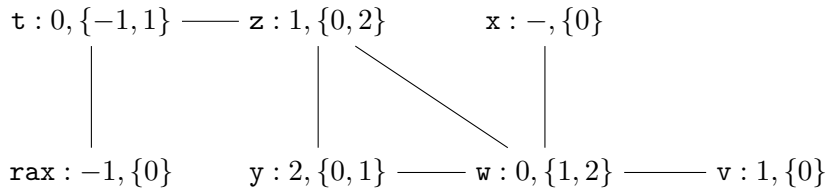
The most saturated vertices are now `w` and `y`. We color `w` with the first available color, which is 0.



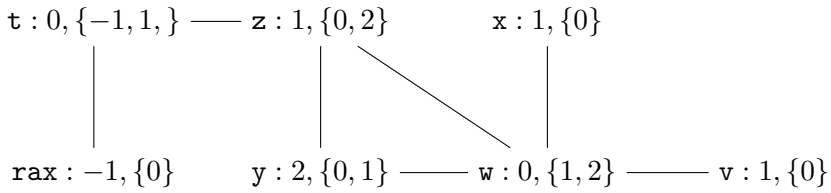
Vertex y is now the most highly saturated, so we color y with 2. We cannot choose 0 or 1 because those numbers are in y's saturation set. Indeed, y interferes with w and z, whose colors are 0 and 1 respectively.



Now x and v are the most saturated, so we color v it 1.



In the last step of the algorithm, we color x with 1.



With the coloring complete, we finalize the assignment of variables to registers and stack locations. Recall that if we have k registers to use for allocation, we map the first k colors to registers and the rest to stack locations. Suppose for the moment that we have just one register to use for register allocation, rcx. Then the following is the mapping of colors to registers and stack allocations.

$$\{0 \mapsto \texttt{\%rcx},\ 1 \mapsto \texttt{-8(\%rbp)},\ 2 \mapsto \texttt{-16(\%rbp)}\}$$

Putting this mapping together with the above coloring of the variables, we arrive at the following assignment of variables to registers and stack locations.

$$\{\texttt{v} \mapsto \texttt{\%rcx}, \texttt{w} \mapsto \texttt{\%rcx}, \texttt{x} \mapsto \texttt{-8(\%rbp)}, \texttt{y} \mapsto \texttt{-16(\%rbp)},$$
$$\texttt{z} \mapsto \texttt{-8(\%rbp)}, \texttt{t} \mapsto \texttt{\%rcx}\}$$

Applying this assignment to our running example, on the left, yields the program on the right.

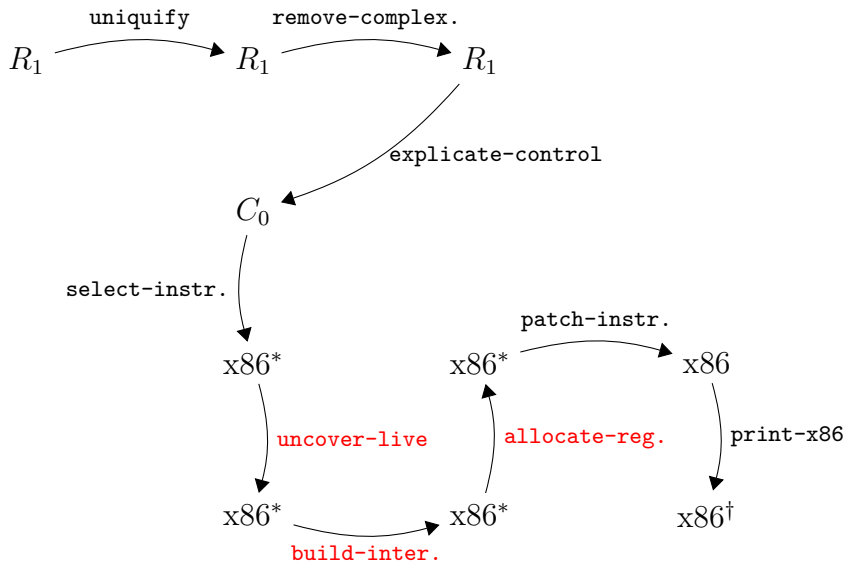


Figure 3.8: Diagram of the passes for R_1 with register allocation.

```
movq $1, v                    movq $1, %rcx
movq $42, w                   movq $42, %rcx
movq v, x                     movq %rcx, -8(%rbp)
addq $7, x                    addq $7, -8(%rbp)
movq x, y                     movq -8(%rbp), -16(%rbp)
movq x, z                ⇒    movq -8(%rbp), -8(%rbp)
addq w, z                     addq %rcx, -8(%rbp)
movq y, t                     movq -16(%rbp), %rcx
negq t                        negq %rcx
movq z, %rax                  movq -8(%rbp), %rax
addq t, %rax                  addq %rcx, %rax
jmp conclusion                jmp conclusion
```

The resulting program is almost an x86 program. The remaining step is the patch instructions pass. In this example, the trivial move of `-8(%rbp)` to itself is deleted and the addition of `-8(%rbp)` to `-16(%rbp)` is fixed by going through `rax` as follows.

```
movq -8(%rbp), %rax
addq %rax, -16(%rbp)
```

An overview of all of the passes involved in register allocation is shown in Figure 3.8.

We recommend creating a helper function named `color-graph` that takes an interference graph and a list of all the variables in the program. This function should return a mapping of variables to their colors (represented as natural numbers). By creating this helper function, you will be able to reuse it in Chapter 6 when you add support for functions. To prioritize the process of highly saturated nodes inside your `color-graph` function, we recommend using the priority queue data structure (see the side bar on the right). Note that you will also need to maintain a mapping from variables to their "handles" in the priority queue so that you can notify the priority queue when their saturation changes.

Once you have obtained the coloring from `color-graph`, you can assign the variables to registers or stack locations and then reuse code from the `assign-homes` pass from Section 2.8 to replace the variables with their assigned location.

Priority Queue

A *priority queue* is a collection of items in which the removal of items is governed by priority. In a "min" queue, lower priority items are removed first. An implementation is in `priority_queue.rkt` of the support code.

`(make-pqueue` *cmp*`)` constructs an empty priority queue that uses the *cmp* predicate to determine whether its first argument has lower or equal priority to its second argument.

`(pqueue-count` *queue*`)` returns the number of items in the queue.

`(pqueue-push!` *queue item*`)` inserts the item into the queue and returns a handle for the item in the queue.

`(pqueue-pop!` *queue*`)` returns the item with the lowest priority.

`(pqueue-decrease-key!` *queue handle*`)` notifices the queue the the priority has decreased for the item associated with the given handle.

Exercise 12. Implement the compiler pass `allocate-registers`, which should come after the `build-interference` pass. The three new passes, `uncover-live`, `build-interference`, and `allocate-registers` replace the `assign-homes` pass of Section 2.8.

Test your updated compiler by creating new example programs that exercise all of the register allocation algorithm, such as forcing variables to be spilled to the stack.

3.5 Print x86 and Conventions for Registers

Recall that the `print-x86` pass generates the prelude and conclusion instructions for the `main` function. The prelude saved the values in `rbp` and `rsp` and the conclusion returned those values to `rbp` and `rsp`. The reason for this is that our `main` function must adhere to the x86 calling conventions that we described in Section 3.1. Furthermore, if your register allocator assigned variables to other callee-saved registers (e.g. `rbx`, `r12`, etc.), then

those variables must also be saved to the stack in the prelude and restored in the conclusion. The simplest approach is to save and restore all of the callee-saved registers. The more efficient approach is to keep track of which callee-saved registers were used and only save and restore them. Either way, make sure to take this use of stack space into account when you are calculating the size of the frame and adjusting the `rsp` in the prelude. Also, don't forget that the size of the frame needs to be a multiple of 16 bytes!

3.6 Challenge: Move Biasing

This section describes an optional enhancement to register allocation for those students who are looking for an extra challenge or who have a deeper interest in register allocation.

We return to the running example, but we remove the supposition that we only have one register to use. So we have the following mapping of color numbers to registers.

$$\{0 \mapsto \texttt{\%rbx},\ 1 \mapsto \texttt{\%rcx},\ 2 \mapsto \texttt{\%rdx}\}$$

Using the same assignment of variables to color numbers that was produced by the register allocator described in the last section, we get the following program.

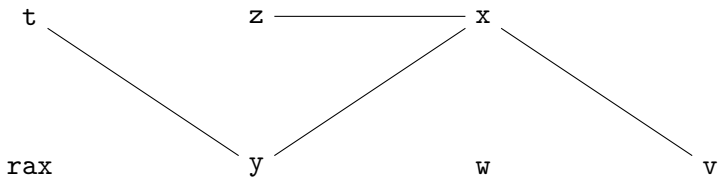
```
movq $1, v                    movq $1, %rcx
movq $42, w                   movq $42, $rbx
movq v, x                     movq %rcx, %rcx
addq $7, x                    addq $7, %rcx
movq x, y                     movq %rcx, %rdx
movq x, z                     movq %rcx, %rcx
addq w, z            ⇒        addq %rbx, %rcx
movq y, t                     movq %rdx, %rbx
negq t                        negq %rbx
movq z, %rax                  movq %rcx, %rax
addq t, %rax                  addq %rbx, %rax
jmp conclusion                jmp conclusion
```

In the above output code there are two `movq` instructions that can be removed because their source and target are the same. However, if we had put `t`, `v`, `x`, and `y` into the same register, we could instead remove three `movq` instructions. We can accomplish this by taking into account which variables appear in `movq` instructions with which other variables.

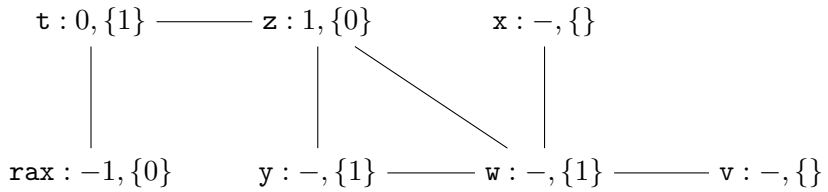
We say that two variables p and q are *move related* if they participate together in a `movq` instruction, that is, `movq` p`,` q or `movq` q`,` p. When the

register allocator chooses a color for a variable, it should prefer a color that has already been used for a move-related variable (assuming that they do not interfere). Of course, this preference should not override the preference for registers over stack locations. This preference should be used as a tie breaker when choosing between registers or when choosing between stack locations.

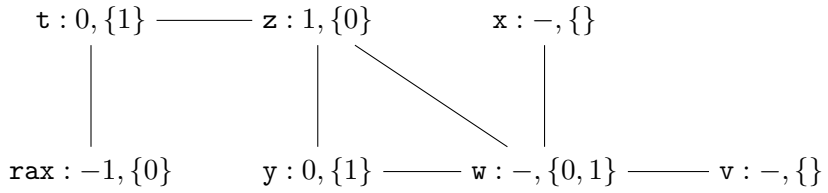
We recommend representing the move relationships in a graph, similar to how we represented interference. The following is the *move graph* for our running example.



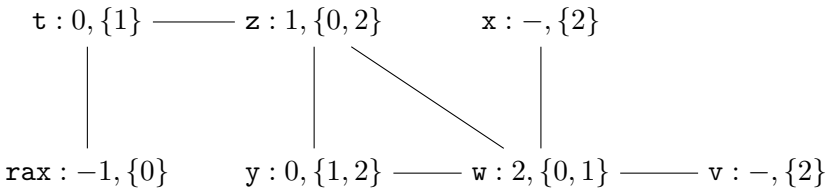
Now we replay the graph coloring, pausing to see the coloring of `y`. Recall the following configuration. The most saturated vertices were `w` and `y`.



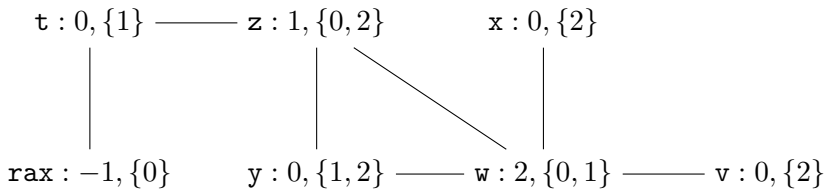
Last time we chose to color `w` with 0. But this time we see that `w` is not move related to any vertex, but `y` is move related to `t`. So we choose to color `y` the same color, 0.



Now `w` is the most saturated, so we color it 2.



At this point, vertices `x` and `v` are most saturated, but `x` is move related to `y` and `z`, so we color `x` to 0 to match `y`. Finally, we color `v` to 0.



So we have the following assignment of variables to registers.

$$\{\texttt{v} \mapsto \texttt{\%rbx}, \texttt{w} \mapsto \texttt{\%rdx}, \texttt{x} \mapsto \texttt{\%rbx}, \texttt{y} \mapsto \texttt{\%rbx}, \texttt{z} \mapsto \texttt{\%rcx}, \texttt{t} \mapsto \texttt{\%rbx}\}$$

We apply this register assignment to the running example, on the left, to obtain the code on right.

```
movq $1, v                    movq $1, %rbx
movq $42, w                   movq $42, %rdx
movq v, x                     movq %rbx, %rbx
addq $7, x                    addq $7, %rbx
movq x, y                     movq %rbx, %rbx
movq x, z                     movq %rbx, %rcx
addq w, z            ⇒        addq %rdx, %rcx
movq y, t                     movq %rbx, %rbx
negq t                        negq %rbx
movq z, %rax                  movq %rcx, %rax
addq t, %rax                  addq %rbx, %rax
jmp conclusion                jmp conclusion
```

The `patch-instructions` then removes the three trivial moves from `rbx` to `rbx` to obtain the following result.

```
movq $1, %rbx
movq $42, %rdx
addq $7, %rbx
movq %rbx, %rcx
addq %rdx, %rcx
negq %rbx
movq %rcx, %rax
addq %rbx, %rax
jmp conclusion
```

Exercise 13. Change your implementation of `allocate-registers` to take move biasing into account. Make sure that your compiler still passes all of

the previous tests. Create two new tests that include at least one opportunity for move biasing and visually inspect the output x86 programs to make sure that your move biasing is working properly.

3.7 Output of the Running Example

Figure 3.9 shows the x86 code generated for the running example (Figure 3.1) with register allocation and move biasing. To demonstrate both the use of registers and the stack, we have limited the register allocator to use just two registers: `rbx` and `rcx`. In the prelude of the `main` function, we push `rbx` onto the stack because it is a callee-saved register and it was assigned to variable by the register allocator. We substract `8` from the `rsp` at the end of the prelude to reserve space for the one spilled variable. After that subtraction, the `rsp` is aligned to 16 bytes.

Moving on the the `start` block, we see how the registers were allocated. Variables `v`, `x`, and `y` were assigned to `rbx` and variable `z` was assigned to `rcx`. Variable `w` was spilled to the stack location `-16(%rbp)`. Recall that the prelude saved the callee-save register `rbx` onto the stack. The spilled variables must be placed lower on the stack than the saved callee-save registers, so in this case `w` is placed at `-16(%rbp)`.

In the `conclusion`, we undo the work that was done in the prelude. We move the stack pointer up by `8` bytes (the room for spilled variables), then we pop the old values of `rbx` and `rbp` (callee-saved registers), and finish with `retq` to return control to the operating system.

```
start:
        movq   $1, %rbx
        movq   $42, -16(%rbp)
        addq   $7, %rbx
        movq   %rbx, %rcx
        addq   -16(%rbp), %rcx
        negq   %rbx
        movq   %rcx, %rax
        addq   %rbx, %rax
        jmp conclusion

        .globl main
main:
        pushq  %rbp
        movq   %rsp, %rbp
        pushq  %rbx
        subq   $8, %rsp
        jmp start

conclusion:
        addq   $8, %rsp
        popq   %rbx
        popq   %rbp
        retq
```

Figure 3.9: The x86 output from the running example (Figure 3.1).

4

Booleans and Control Flow

The R_0 and R_1 languages only have a single kind of value, the integers. In this chapter we add a second kind of value, the Booleans, to create the R_2 language. The Boolean values *true* and *false* are written `#t` and `#f` respectively in Racket. The R_2 language includes several operations that involve Booleans (`and`, `not`, `eq?`, `<`, etc.) and the conditional `if` expression. With the addition of `if` expressions, programs can have non-trivial control flow which which significantly impacts the `explicate-control` and the liveness analysis for register allocation. Also, because we now have two kinds of values, we need to handle programs that apply an operation to the wrong kind of value, such as `(not 1)`.

There are two language design options for such situations. One option is to signal an error and the other is to provide a wider interpretation of the operation. The Racket language uses a mixture of these two options, depending on the operation and the kind of value. For example, the result of `(not 1)` in Racket is `#f` because Racket treats non-zero integers as if they were `#t`. On the other hand, `(car 1)` results in a run-time error in Racket stating that `car` expects a pair.

The Typed Racket language makes similar design choices as Racket, except much of the error detection happens at compile time instead of run time. Like Racket, Typed Racket accepts and runs `(not 1)`, producing `#f`. But in the case of `(car 1)`, Typed Racket reports a compile-time error because Typed Racket expects the type of the argument to be of the form `(Listof T)` or `(Pairof T1 T2)`.

For the R_2 language we choose to be more like Typed Racket in that we shall perform type checking during compilation. In Chapter 8 we study the alternative choice, that is, how to compile a dynamically typed language

bool	::=	`#t` | `#f`
cmp	::=	`eq?` | `<` | `<=` | `>` | `>=`
exp	::=	*int* | (`read`) | (`-` *exp*) | (`+` *exp exp*) | (`-` *exp exp*)
	|	*var* | (`let` ([*var exp*]) *exp*)
	|	*bool* | (`and` *exp exp*) | (`or` *exp exp*) | (`not` *exp*)
	|	(*cmp exp exp*) | (`if` *exp exp exp*)
R_2	::=	*exp*

Figure 4.1: The concrete syntax of R_2, extending R_1 (Figure 2.1) with Booleans and conditionals.

like Racket. The R_2 language is a subset of Typed Racket but by no means includes all of Typed Racket. For many operations we take a narrower interpretation than Typed Racket, for example, rejecting `(not 1)`.

This chapter is organized as follows. We begin by defining the syntax and interpreter for the R_2 language (Section 4.1). We then introduce the idea of type checking and build a type checker for R_2 (Section 4.2). To compile R_2 we need to enlarge the intermediate language C_0 into C_1, which we do in Section 4.5. The remaining sections of this chapter discuss how our compiler passes need to change to accommodate Booleans and conditional control flow.

4.1 The R_2 Language

The concrete syntax of the R_2 language is defined in Figure 4.1 and the abstract syntax is defined in Figure 4.2. The R_2 language includes all of R_1 (shown in gray), the Boolean literals `#t` and `#f`, and the conditional `if` expression. Also, we expand the operators to include

1. subtraction on integers,
2. the logical operators `and`, `or` and `not`,
3. the `eq?` operation for comparing two integers or two Booleans, and
4. the `<`, `<=`, `>`, and `>=` operations for comparing integers.

Figure 4.3 defines the interpreter for R_2, omitting the parts that are the same as the interpreter for R_1 (Figure 2.3). The literals `#t` and `#f` evaluate to the corresponding Boolean values. The conditional expression

```
bool  ::= #t | #f
cmp   ::= eq? | < | <= | > | >=
exp   ::= (Int int) | (Prim 'read '())
        | (Prim '- (list exp )) | (Prim '+ (list exp exp))
        | (Prim '- (list exp exp))
        | (Var var) | (Let var exp exp)
        | (Bool bool) | (Prim 'and (list exp exp))
        | (Prim 'or (list exp exp)) | (Prim 'not (list exp ))
        | (Prim cmp (list exp exp)) | (If exp exp exp)
R2    ::= (Program '() exp)
```

Figure 4.2: The abstract syntax of R_2.

(`if` *cnd thn els*) evaluates the Boolean expression *cnd* and then either evaluates *thn* or *els* depending on whether *cnd* produced `#t` or `#f`. The logical operations `not` and `and` behave as you might expect, but note that the `and` operation is short-circuiting. That is, given the expression (`and` e_1 e_2), the expression e_2 is not evaluated if e_1 evaluates to `#f`.

With the addition of the comparison operations, there are quite a few primitive operations and the interpreter code for them could become repetitive without some care. In Figure 4.3 we factor out the different parts of the code for primitive operations into the `interp-op` function and the similar parts of the code into the match clause for `Prim` shown in Figure 4.3. We do not use `interp-op` for the `and` operation because of the short-circuiting behavior in the order of evaluation of its arguments.

4.2 Type Checking R_2 Programs

It is helpful to think about type checking in two complementary ways. A type checker predicts the type of value that will be produced by each expression in the program. For R_2, we have just two types, `Integer` and `Boolean`. So a type checker should predict that

```
(+ 10 (- (+ 12 20)))
```

produces an `Integer` while

```
(and (not #f) #t)
```

produces a `Boolean`.

```
(define (interp-op op)
  (match op
    ...
    ['not (lambda (v) (match v [#t #f] [#f #t]))]
    ['eq? (lambda (v1 v2)
            (cond [(or (and (fixnum? v1) (fixnum? v2))
                       (and (boolean? v1) (boolean? v2)))
                   (eq? v1 v2)]))]
    ['< (lambda (v1 v2)
          (cond [(and (fixnum? v1) (fixnum? v2)) (< v1 v2)]))]
    ['<= (lambda (v1 v2)
           (cond [(and (fixnum? v1) (fixnum? v2)) (<= v1 v2)]))]
    ['> (lambda (v1 v2)
          (cond [(and (fixnum? v1) (fixnum? v2)) (> v1 v2)]))]
    ['>= (lambda (v1 v2)
           (cond [(and (fixnum? v1) (fixnum? v2)) (>= v1 v2)]))]
    [else (error 'interp-op "unknown operator")]))

(define (interp-exp env)
  (lambda (e)
    (define recur (interp-exp env))
    (match e
      ...
      [(Bool b) b]
      [(If cnd thn els)
       (define b (recur cnd))
       (match b
         [#t (recur thn)]
         [#f (recur els)])]
      [(Prim 'and (list e1 e2))
       (define v1 (recur e1))
       (match v1
         [#t (match (recur e2) [#t #t] [#f #f])]
         [#f #f])]
      [(Prim op args)
       (apply (interp-op op) (for/list ([e args]) (recur e)))]
      )))

(define (interp-R2 p)
  (match p
    [(Program info e)
     ((interp-exp '()) e)]
    ))
```

Figure 4.3: Interpreter for the R_2 language.

Another way to think about type checking is that it enforces a set of rules about which operators can be applied to which kinds of values. For example, our type checker for R_2 will signal an error for the below expression because, as we have seen above, the expression `(+ 10 ...)` has type `Integer` but the type checker enforces the rule that the argument of `not` must be a `Boolean`.

```
(not (+ 10 (- (+ 12 20))))
```

The type checker for R_2 is a structurally recursive function over the AST. Figure 4.4 shows many of the clauses for the `type-check-exp` function. Given an input expression `e`, the type checker either returns a type (`Integer` or `Boolean`) or it signals an error. The type of an integer literal is `Integer` and the type of a Boolean literal is `Boolean`. To handle variables, the type checker uses an environment that maps variables to types. Consider the clause for `let`. We type check the initializing expression to obtain its type `T` and then associate type `T` with the variable `x` in the environment. When the type checker encounters a use of variable `x` in the body of the `let`, it can find its type in the environment.

Exercise 14. Complete the implementation of `type-check`. Test your type checker using `interp-tests` and `compiler-tests` by passing the `type-check` function as the second argument. Create 10 new example programs in R_2 that you choose based on how thoroughly they test you type checking function. Half of the example programs should have a type error to make sure that your type checker properly rejects them. For those programs, to signal that a type error is expected, create an empty file with the same base name but with file extension `.tyerr`. For example, if the test `r2_14.rkt` is expected to error, then create an empty file named `r2_14.tyerr`. The other half of the example programs should not have type errors. Note that if your type checker does not signal an error for a program, then interpreting that program should not encounter an error. If it does, there is something wrong with your type checker.

4.3 Shrink the R_2 Language

The R_2 language includes several operators that are easily expressible in terms of other operators. For example, subtraction is expressible in terms of addition and negation.

$$\texttt{(-}\ e_1\ e_2\texttt{)} \quad \Rightarrow \quad \texttt{(+}\ e_1\ \texttt{(-}\ e_2\texttt{))}$$

```
(define (type-check-exp env)
  (lambda (e)
    (match e
      [(Var x) (dict-ref env x)]
      [(Int n) 'Integer]
      [(Bool b) 'Boolean]
      [(Let x e body)
        (define Te ((type-check-exp env) e))
        (define Tb ((type-check-exp (dict-set env x Te)) body))
        Tb]
      ...
      [else
       (error "type-check-exp couldn't match" e)])))

(define (type-check env)
  (lambda (e)
    (match e
      [(Program info body)
       (define Tb ((type-check-exp '()) body))
       (unless (equal? Tb 'Integer)
         (error "result of the program must be an integer, not " Tb))
       (Program info body)]
      )))
```

Figure 4.4: Skeleton of a type checker for the R_2 language.

Several of the comparison operations are expressible in terms of less-than and logical negation.

$$\texttt{(<=}\ e_1\ e_2\texttt{)} \quad \Rightarrow \quad \texttt{(let ([tmp.1}\ e_1\texttt{]) (not (<}\ e_2\ \texttt{tmp.1)))}$$

The `let` is needed in the above translation to ensure that expression e_1 is evaluated before e_2.

By performing these translations near the front-end of the compiler, the later passes of the compiler do not need to deal with these constructs, making those passes shorter. On the other hand, sometimes these translations make it more difficult to generate the most efficient code with respect to the number of instructions. However, these differences typically do not affect the number of accesses to memory, which is the primary factor that determines execution time on modern computer architectures.

Exercise 15. Implement the pass `shrink` that removes subtraction, `and`, `or`, `<=`, `>`, and `>=` from the language by translating them to other constructs in R_2. Create tests to make sure that the behavior of all of these constructs stays the same after translation.

4.4 The x86$_1$ Language

To implement the new logical operations, the comparison operations, and the `if` expression, we need to delve further into the x86 language. Figures 4.5 and 4.6 define the concrete and abstract syntax for a larger subset of x86 that includes instructions for logical operations, comparisons, and conditional jumps.

One small challenge is that x86 does not provide an instruction that directly implements logical negation (`not` in R_2 and C_1). However, the `xorq` instruction can be used to encode `not`. The `xorq` instruction takes two arguments, performs a pairwise exclusive-or (XOR) operation on each bit of its arguments, and writes the results into its second argument. Recall the truth table for exclusive-or:

	0	1
0	0	1
1	1	0

For example, applying XOR to each bit of the binary numbers 0011 and 0101 yields 0110. Notice that in the row of the table for the bit 1, the result is

```
bytereg  ::= ah | al | bh | bl | ch | cl | dh | dl
arg      ::= $int | %reg | int(%reg) | %bytereg
cc       ::= e | l | le | g | ge
instr    ::= addq arg,arg | subq arg,arg | negq arg | movq arg,arg |
             callq label | pushq arg | popq arg | retq | jmp label
             label: instr | xorq arg, arg | cmpq arg, arg |
             setcc arg | movzbq arg, arg | jcc label
x86_1    ::= .globl main
             main: instr ...
```

Figure 4.5: The concrete syntax of $x86_1$ (extends $x86_0$ of Figure 2.4).

```
bytereg  ::= ah | al | bh | bl | ch | cl | dh | dl
arg      ::= (Imm int) | (Reg reg) | (Deref reg int) | (ByteReg bytereg)
cc       ::= e | l | le | g | ge
instr    ::= (Instr 'addq (list arg arg)) | (Instr 'subq (list arg arg))
           | (Instr 'movq (list arg arg)) | (Instr 'negq (list arg))
           | (Callq label) | (Retq) | (Pushq arg) | (Popq arg) | (Jmp label)
           | (Instr 'xorq (list arg arg)) | (Instr 'cmpq (list arg arg))
           | (Instr 'set (list cc arg)) | (Instr 'movzbq (list arg arg))
           | (JmpIf cc label)
block    ::= (Block info instr ...)
x86_1    ::= (Program info (CFG ((label . block) ...)))
```

Figure 4.6: The abstract syntax of $x86_1$ (extends $x86_0$ of Figure 2.8).

the opposite of the second bit. Thus, the `not` operation can be implemented by `xorq` with 1 as the first argument:

$$\texttt{(not } arg\texttt{);} \quad \Rightarrow \quad \begin{array}{l} \texttt{movq } arg\texttt{,}var \\ \texttt{xorq \$1,}var \end{array}$$

Next we consider the x86 instructions that are relevant for compiling the comparison operations. The `cmpq` instruction compares its two arguments to determine whether one argument is less than, equal, or greater than the other argument. The `cmpq` instruction is unusual regarding the order of its arguments and where the result is placed. The argument order is backwards: if you want to test whether $x < y$, then write `cmpq` y`,` x. The result of `cmpq` is placed in the special EFLAGS register. This register cannot be accessed directly but it can be queried by a number of instructions, including the `set`

instruction. The `set` instruction puts a `1` or `0` into its destination depending on whether the comparison came out according to the condition code *cc* (`e` for equal, `l` for less, `le` for less-or-equal, `g` for greater, `ge` for greater-or-equal). The `set` instruction has an annoying quirk in that its destination argument must be single byte register, such as `al` (L for lower bits) or `ah` (H for higher bits), which are part of the `rax` register. Thankfully, the `movzbq` instruction can then be used to move from a single byte register to a normal 64-bit register.

The x86 instruction for conditional jump are relevant to the compilation of `if` expressions. The `JmpIf` instruction updates the program counter to point to the instruction after the indicated label depending on whether the result in the EFLAGS register matches the condition code *cc*, otherwise the `JmpIf` instruction falls through to the next instruction. The abstract syntax for `JmpIf` differs from the concrete syntax for x86 in that it separates the instruction name from the condition code. For example, `(JmpIf le foo)` corresponds to `jle foo`. Because the `JmpIf` instruction relies on the EFLAGS register, it is common for the `JmpIf` to be immediately preceded by a `cmpq` instruction to set the EFLAGS register.

4.5 The C_1 Intermediate Language

As with R_1, we compile R_2 to a C-like intermediate language, but we need to grow that intermediate language to handle the new features in R_2: Booleans and conditional expressions. Figure 4.7 defines the concrete syntax of C_1 and Figure 4.8 defines the abstract syntax. In particular, we add logical and comparison operators to the *exp* non-terminal and the literals `#t` and `#f` to the *arg* non-terminal. Regarding control flow, C_1 differs considerably from R_2. Instead of `if` expressions, C_1 has `goto` and conditional `goto` in the grammar for *tail*. This means that a sequence of statements may now end with a `goto` or a conditional `goto`. The conditional `goto` jumps to one of two labels depending on the outcome of the comparison. In Section 4.7 we discuss how to translate from R_2 to C_1, bridging this gap between `if` expressions and `goto`'s.

```
atm    ::=  int | var | bool
cmp    ::=  eq? | <
exp    ::=  atm | (read) | (- atm) | (+ atm atm)
          |  (not atm) | (cmp atm atm)
stmt   ::=  var = exp;
tail   ::=  return exp; | stmt tail | goto label;
          |  if (cmp atm atm) goto label; else goto label;
C1     ::=  (label: tail) ...
```

Figure 4.7: The concrete syntax of the C_1 intermediate language.

```
atm    ::=  (Int int) | (Var var) | (Bool bool)
cmp    ::=  eq? | <
exp    ::=  atm | (Prim 'read '())
          |  (Prim '- (list atm )) | (Prim '+ (list atm atm))
          |  (Prim 'not (list atm)) | (Prim 'cmp (list atm atm))
stmt   ::=  (Assign (Var var) exp)
tail   ::=  (Return exp) | (Seq stmt tail) | (Goto label)
          |  (IfStmt (Prim cmp (list atm atm)) (Goto label) (Goto label))
C1     ::=  (Program info (CFG (label . tail) ...))
```

Figure 4.8: The abstract syntax of C_1, an extention of C_0 (Figure 2.11).

$$
\begin{array}{rcl}
\mathit{atm} & ::= & \texttt{(Int } \mathit{int}\texttt{)} \mid \texttt{(Var } \mathit{var}\texttt{)} \mid \texttt{(Bool } \mathit{bool}\texttt{)} \\
\mathit{exp} & ::= & \mathit{atm} \mid \texttt{(Prim 'read '())} \\
 & \mid & \texttt{(Prim '- (list } \mathit{atm}\texttt{))} \mid \texttt{(Prim '+ (list } \mathit{atm}\ \mathit{atm}\texttt{))} \\
 & \mid & \texttt{(Let } \mathit{var}\ \mathit{exp}\ \mathit{exp}\texttt{)} \\
 & \mid & \texttt{(Prim 'not (list } \mathit{atm}\texttt{))} \\
 & \mid & \texttt{(Prim } \mathit{cmp}\ \texttt{(list } \mathit{atm}\ \mathit{atm}\texttt{))} \mid \texttt{(If } \mathit{exp}\ \mathit{exp}\ \mathit{exp}\texttt{)} \\
R_2^\dagger & ::= & \texttt{(Program '() } \mathit{exp}\texttt{)}
\end{array}
$$

Figure 4.9: $R_2^\dagger$ is R_2 in administrative normal form (ANF).

4.6 Remove Complex Operands

Add cases for `Bool` and `If` to the `rco-exp` and `rco-atom` functions according to the definition of the output language for this pass, $R_2^\dagger$, the administrative normal form of R_2, which is defined in Figure 4.9. The `Bool` form is an atomic expressions but `If` is not. All three sub-expressions of an `If` are allowed to be complex expressions in the output of `remove-complex-opera*`, but the operands of `not` and the comparisons must be atoms. Regarding the `If` form, it is particularly important to **not** replace its condition with a temporary variable because that would interfere with the generation of high-quality output in the `explicate-control` pass.

4.7 Explicate Control

Recall that the purpose of `explicate-control` is to make the order of evaluation explicit in the syntax of the program. With the addition of `if` in R_2 this get more interesting.

As a motivating example, consider the following program that has an `if` expression nested in the predicate of another `if`.

```
(let ([x (read)])
  (let ([y (read)])
    (if (if (< x 1) (eq? x 0) (eq? x 2))
        (+ y 2)
        (+ y 10))))
```

The naive way to compile `if` and the comparison would be to handle each of them in isolation, regardless of their context. Each comparison would be

translated into a `cmpq` instruction followed by a couple instructions to move the result from the EFLAGS register into a general purpose register or stack location. Each `if` would be translated into the combination of a `cmpq` and a conditional jump. The generated code for the inner `if` in the above example would be as follows.

```
...
cmpq $1, x          ;; (< x 1)
setl %al
movzbq %al, tmp
cmpq $1, tmp        ;; (if (< x 1) ...)
je then_branch_1
jmp else_branch_1
...
```

However, if we take context into account we can do better and reduce the use of `cmpq` and EFLAG-accessing instructions.

One idea is to try and reorganize the code at the level of R_2, pushing the outer `if` inside the inner one. This would yield the following code.

```
(let ([x (read)])
  (let ([y (read)])
    (if (< x 1)
      (if (eq? x 0)
        (+ y 2)
        (+ y 10))
      (if (eq? x 2)
        (+ y 2)
        (+ y 10)))))
```

Unfortunately, this approach duplicates the two branches, and a compiler must never duplicate code!

We need a way to perform the above transformation, but without duplicating code. The solution is straightforward if we think at the level of x86 assembly: we can label the code for each of the branches and insert jumps in all the places that need to execute the branches. Put another way, we need to move away from abstract syntax *trees* and instead use *graphs*. In particular, we shall use a standard program representation called a *control flow graph* (CFG), due to Frances Elizabeth Allen [1970]. Each vertex is a labeled sequence of code, called a *basic block*, and each edge represents a jump to another block. The `Program` construct of C_0 and C_1 contains a control flow graph represented as an alist mapping labels to basic blocks. Each basic block is represented by the *tail* non-terminal.

Figure 4.10 shows the output of the `remove-complex-opera*` pass and then the `explicate-control` pass on the example program. We walk through the output program and then discuss the algorithm. Following the order of evaluation in the output of `remove-complex-opera*`, we first have two calls to `(read)` and then the less-than-comparison to `1` in the predicate of the inner `if`. In the output of `explicate-control`, in the block labeled `start`, this becomes two assignment statements followed by a conditional `goto` to label `block96` or `block97`. The blocks associated with those labels contain the translations of the code `(eq? x 0)` and `(eq? x 2)`, respectively. Regarding the block labeled with `block96`, we start with the comparison to `0` and then have a conditional goto, either to label `block92` or label `block93`, which indirectly take us to labels `block90` and `block91`, the two branches of the outer `if`, i.e., `(+ y 2)` and `(+ y 10)`. The story for the block labeled `block97` is similar.

The nice thing about the output of `explicate-control` is that there are no unnecessary comparisons and every comparison is part of a conditional jump. The down-side of this output is that it includes trivial blocks, such as the blocks labeled `block92` through `block95`, that only jump to another block. We discuss a solution to this problem in Section 4.12.

Recall that in Section 2.6 we implement `explicate-control` for R_1 using two mutually recursive functions, `explicate-tail` and `explicate-assign`. The former function translates expressions in tail position whereas the later function translates expressions on the right-hand-side of a `let`. With the addition of `if` expression in R_2 we have a new kind of context to deal with: the predicate position of the `if`. We need another function, `explicate-pred`, that takes an R_2 expression and two blocks (two C_1 *tail* AST nodes) for the then-branch and else-branch. The output of `explicate-pred` is a block and a list of formerly `let`-bound variables.

Note that the three explicate functions need to construct a control-flow graph, which we recommend they do via updates to a global variable.

In the following paragraphs we consider the specific additions to the `explicate-tail` and `explicate-assign` functions, and some of cases for the `explicate-pred` function.

The `explicate-tail` function needs an additional case for `if`. The branches of the `if` inherit the current context, so they are in tail position. Let B_1 be the result of `explicate-tail` on the "then" branch of the `if`, so B_1 is a *tail* AST node. Let B_2 be the result of apply `explicate-tail` to the "else" branch. Finally, let B_3 be the *tail* that results fromapplying `explicate-pred` to the predicate *cnd* and the blocks B_1 and B_2. Then the

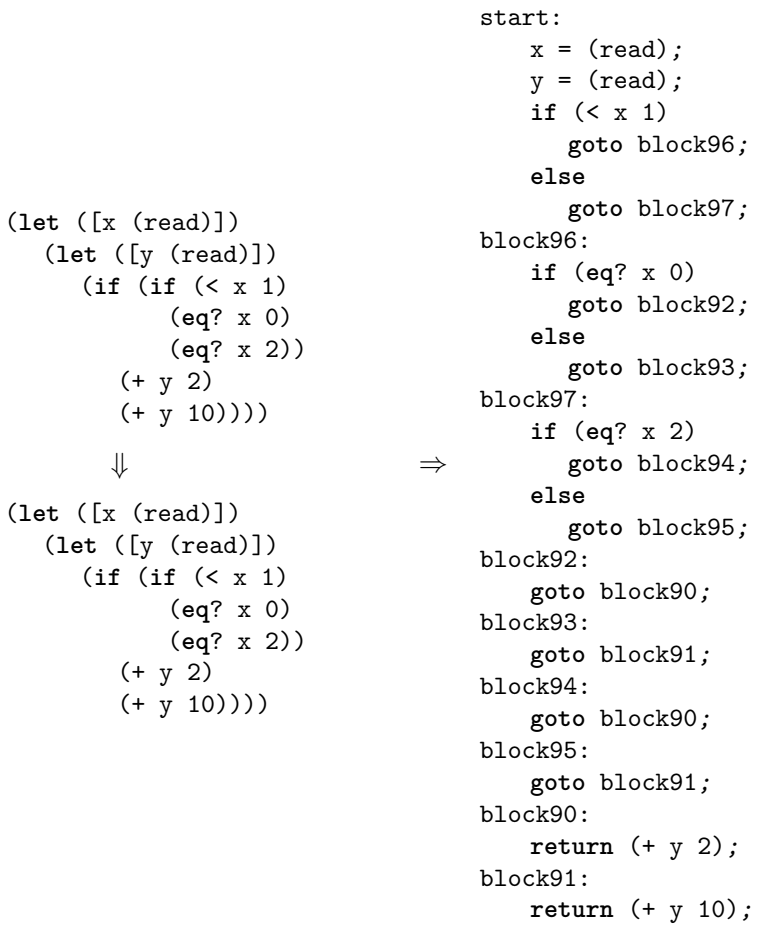


Figure 4.10: Example translation from R_2 to C_1 via the `explicate-control`.

`if` as a whole translates to block B_3.

$$(\texttt{if}\ \mathit{cnd}\ \mathit{thn}\ \mathit{els}) \quad \Rightarrow \quad B_3$$

In the above discussion, we use the metavariables B_1, B_2, and B_3 to refer to blocks for the purposes of our discussion, but they should not be confused with the labels for the blocks that appear in the generated code. We initially construct unlabeled blocks; we only attach labels to blocks when we add them to the control-flow graph, as we shall see in the next case.

Next consider the case for `if` in the `explicate-assign` function. The context of the `if` is an assignment to some variable x and then the control continues to some block B_1. The code that we generate for both the "then" and "else" branches needs to continue to B_1, so to avoid duplicating B_1 we instead add it to the control flow graph with a fresh label ℓ_1. The branches of the `if` inherit the current context, so that are in assignment positions. Let B_2 be the result of applying `explicate-assign` to the "then" branch, variable x, and the block `(Goto` ℓ_1`)`. Let B_3 be the result of applying `explicate-assign` to the "else" branch, variable x, and the block `(Goto` ℓ_1`)`. Finally, let B_4 be the result of applying `explicate-pred` to the predicate *cnd* and the blocks B_2 and B_3. The `if` as a whole translates to the block B_4.

$$(\texttt{if}\ \mathit{cnd}\ \mathit{thn}\ \mathit{els}) \quad \Rightarrow \quad B_4$$

The function `explicate-pred` will need a case for every expression that can have type `Boolean`. We detail a few cases here and leave the rest for the reader. The input to this function is an expression and two blocks, B_1 and B_2, for the two branches of the enclosing `if`. Suppose the expression is the Boolean `#t`. Then we can perform a kind of partial evaluation and translate it to the "then" branch B_1. Likewise, we translate `#f` to the "else" branch B_2.

$$\texttt{\#t} \quad \Rightarrow \quad B_1, \qquad\qquad \texttt{\#f} \quad \Rightarrow \quad B_2$$

Next, suppose the expression is a less-than comparison. We translate it to a conditional `goto`. We need labels for the two branches B_1 and B_2, so we add those blocks to the control flow graph and obtain their labels ℓ_1 and ℓ_2. The translation of the less-than comparison is as follows.

$$(\texttt{<}\ e_1\ e_2) \quad \Rightarrow \quad \begin{array}{l} \texttt{if}\ (\texttt{<}\ e_1\ e_2) \\ \quad \texttt{goto}\ \ell_1\texttt{;} \\ \texttt{else} \\ \quad \texttt{goto}\ \ell_2\texttt{;} \end{array}$$

The case for `if` in `explicate-pred` is particularly illuminating as it deals with the challenges that we discussed above regarding the example of the nested `if` expressions. Again, we add the two branches B_1 and B_2 to the control flow graph and obtain their labels ℓ_1 and ℓ_2. The "then" and "else" branches of the current `if` inherit their context from the current one, that is, predicate context. So we apply `explicate-pred` to the "then" branch with the two blocks `(Goto` ℓ_1`)` and `(Goto` ℓ_2`)` to obtain B_3. Proceed in a similar way with the "else" branch to obtain B_4. Finally, we apply `explicate-pred` to the predicate of the `if` and the blocks B_3 and B_4 to obtain the result B_5.

$$(\texttt{if}\ \mathit{cnd}\ \mathit{thn}\ \mathit{els}) \quad \Rightarrow \quad B_5$$

Finally, note that the way in which the `shrink` pass transforms logical operations such as `and` and `or` can impact the quality of code generated by `explicate-control`. For example, consider the following program.

```
(if (and (eq? (read) 0) (eq? (read) 1))
    0
    42)
```

The `and` operation should transform into something that the `explicat-pred` function can still analyze and descend through to reach the underlying `eq?` conditions. Ideally, your `explicate-control` pass should generate code similar to the following for the above program.[1]

```
start:
    tmp13 = (read);
    if (eq? tmp13 0)
       goto block19;
    else
       goto block20;
block19:
    tmp14 = (read);
    if (eq? tmp14 1)
       goto block17;
    else
       goto block18;
block20:
    goto block16;
block17:
    goto block15;
block18:
    goto block16;
block15:
    return 0;
block16:
    return 42;
```

Exercise 16. Implement the pass `explicate-control` by adding the cases for `if` to the functions for tail and assignment contexts, and implement `explicate-pred` for predicate contexts. Create test cases that exercise all of the new cases in the code for this pass.

[1] If the trivial blocks 17, 18, and 20 bother you, take a look at the challenge problem in Section 4.12.

4.8 Select Instructions

Recall that the `select-instructions` pass lowers from our *C*-like intermediate representation to the pseudo-x86 language, which is suitable for conducting register allocation. The pass is implemented using three auxiliary functions, one for each of the non-terminals *atm*, *stmt*, and *tail*.

For *atm*, we have new cases for the Booleans. We take the usual approach of encoding them as integers, with true as 1 and false as 0.

$$\texttt{\#t} \Rightarrow \texttt{1} \qquad \texttt{\#f} \Rightarrow \texttt{0}$$

For *stmt*, we discuss a couple cases. The `not` operation can be implemented in terms of `xorq` as we discussed at the beginning of this section. Given an assignment *var* `= (not` *atm*`);`, if the left-hand side *var* is the same as *atm*, then just the `xorq` suffices.

$$\textit{var}\ \texttt{= (not}\ \textit{var}\texttt{);} \quad \Rightarrow \quad \texttt{xorq \$1,}\ \textit{var}$$

Otherwise, a `movq` is needed to adapt to the update-in-place semantics of x86. Let *arg* be the result of translating *atm* to x86. Then we have

$$\textit{var}\ \texttt{= (not}\ \textit{atm}\texttt{);} \quad \Rightarrow \quad \begin{array}{l}\texttt{movq}\ \textit{arg}\texttt{,}\ \textit{var}\\ \texttt{xorq \$1,}\ \textit{var}\end{array}$$

Next consider the cases for `eq?` and less-than comparison. Translating these operations to x86 is slightly involved due to the unusual nature of the `cmpq` instruction discussed above. We recommend translating an assignment from `eq?` into the following sequence of three instructions.

$$\textit{var}\ \texttt{= (eq?}\ \textit{atm}_1\ \textit{atm}_2\texttt{);} \quad \Rightarrow \quad \begin{array}{l}\texttt{cmpq}\ \textit{arg}_2\texttt{,}\ \textit{arg}_1\\ \texttt{sete \%al}\\ \texttt{movzbq \%al,}\ \textit{var}\end{array}$$

Regarding the *tail* non-terminal, we have two new cases: `goto` and conditional `goto`. Both are straightforward to handle. A `goto` becomes a jump instruction.

$$\texttt{goto}\ \ell\texttt{;} \quad \Rightarrow \quad \texttt{jmp}\ \ell$$

A conditional `goto` becomes a compare instruction followed by a conditional jump (for "then") and the fall-through is to a regular jump (for "else").

$$\begin{array}{l}\texttt{if (eq?}\ \textit{atm}_1\ \textit{atm}_2\texttt{)}\\ \quad \texttt{goto}\ \ell_1\texttt{;}\\ \texttt{else}\\ \quad \texttt{goto}\ \ell_2\texttt{;}\end{array} \quad \Rightarrow \quad \begin{array}{l}\texttt{cmpq}\ \textit{arg}_2\texttt{,}\ \textit{arg}_1\\ \texttt{je}\ \ell_1\\ \texttt{jmp}\ \ell_2\end{array}$$

Exercise 17. Expand your `select-instructions` pass to handle the new features of the R_2 language. Test the pass on all the examples you have created and make sure that you have some test programs that use the `eq?` and `<` operators, creating some if necessary. Test the output using the `interp-x86` interpreter (Appendix 12.1).

4.9 Register Allocation

The changes required for R_2 affect liveness analysis, building the interference graph, and assigning homes, but the graph coloring algorithm itself does not change.

4.9.1 Liveness Analysis

Recall that for R_1 we implemented liveness analysis for a single basic block (Section 3.2). With the addition of `if` expressions to R_2, `explicate-control` produces many basic blocks arranged in a control-flow graph. The first question we need to consider is: what order should we process the basic blocks? Recall that to perform liveness analysis, we need to know the live-after set. If a basic block has no successor blocks (i.e. no out-edges in the control flow graph), then it has an empty live-after set and we can immediately apply liveness analysis to it. If a basic block has some successors, then we need to complete liveness analysis on those blocks first. Furthermore, we know that the control flow graph does not contain any cycles because R_2 does not include loops [2]. Returning to the question of what order should we process the basic blocks, the answer is reverse topological order. We recommend using the `tsort` (topological sort) and `transpose` functions of the Racket `graph` package to obtain this ordering.

The next question is how to compute the live-after set of a block given the live-before sets of all its successor blocks. (There can be more than one because of conditional jumps.) During compilation we do not know which way a conditional jump will go, so we do not know which of the successor's live-before set to use. The solution to this challenge is based on the observation that there is no harm to the correctness of the compiler if we classify more variables as live than the ones that are truly live during a particular execution of the block. Thus, we can take the union of the

[2] If we were to add loops to the language, then the CFG could contain cycles and we would instead need to use the classic worklist algorithm for computing the fixed point of the liveness analysis [Aho et al., 1986].

live-before sets from all the successors to be the live-after set for the block. Once we have computed the live-after set, we can proceed to perform liveness analysis on the block just as we did in Section 3.2.

The helper functions for computing the variables in an instruction's argument and for computing the variables read-from (R) or written-to (W) by an instruction need to be updated to handle the new kinds of arguments and instructions in x86_1.

4.9.2 Build Interference

Many of the new instructions in x86_1 can be handled in the same way as the instructions in x86_0. Thus, if your code was already quite general, it will not need to be changed to handle the new instructions. If you code is not general enough, I recommend that you change your code to be more general. For example, you can factor out the computing of the the read and write sets for each kind of instruction into two auxiliary functions.

Note that the `movzbq` instruction requires some special care, just like the `movq` instruction. See rule number 3 in Section 3.3.

Exercise 18. Update the `register-allocation` pass so that it works for R_2 and test your compiler using your previously created programs on the `interp-x86` interpreter (Appendix 12.1).

4.10 Patch Instructions

The second argument of the `cmpq` instruction must not be an immediate value (such as an integer). So if you are comparing two immediates, we recommend inserting a `movq` instruction to put the second argument in `rax`. The second argument of the `movzbq` must be a register. There are no special restrictions on the x86 instructions `JmpIf` and `Jmp`.

Exercise 19. Update `patch-instructions` to handle the new x86 instructions. Test your compiler using your previously created programs on the `interp-x86` interpreter (Appendix 12.1).

4.11 An Example Translation

Figure 4.11 shows a simple example program in R_2 translated to x86, showing the results of `explicate-control`, `select-instructions`, and the final x86 assembly code.

Figure 4.12 lists all the passes needed for the compilation of R_2.

```
(if (eq? (read) 1) 42 0)
```

⇓

```
start:
    tmp7951 = (read);
    if (eq? tmp7951 1) then
       goto block7952;
    else
       goto block7953;
block7952:
    return 42;
block7953:
    return 0;
```

⇓

```
start:
    callq read_int
    movq %rax, tmp7951
    cmpq $1, tmp7951
    je block7952
    jmp block7953
block7953:
    movq $0, %rax
    jmp conclusion
block7952:
    movq $42, %rax
    jmp conclusion
```

⇒

```
start:
       callq  read_int
       movq   %rax, %rcx
       cmpq   $1, %rcx
       je block7952
       jmp block7953
block7953:
       movq   $0, %rax
       jmp conclusion
block7952:
       movq   $42, %rax
       jmp conclusion

       .globl main
main:
       pushq  %rbp
       movq   %rsp, %rbp
       pushq  %r13
       pushq  %r12
       pushq  %rbx
       pushq  %r14
       subq   $0, %rsp
       jmp start
conclusion:
       addq   $0, %rsp
       popq   %r14
       popq   %rbx
       popq   %r12
       popq   %r13
       popq   %rbp
       retq
```

Figure 4.11: Example compilation of an `if` expression to x86.

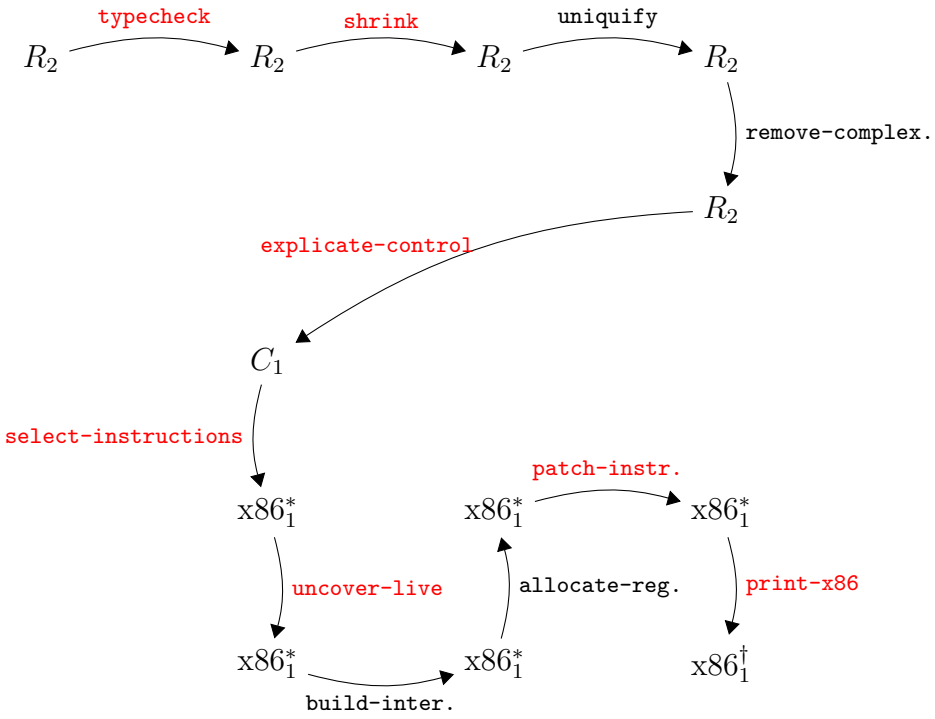


Figure 4.12: Diagram of the passes for R_2, a language with conditionals.

4.12 Challenge: Optimize and Remove Jumps

Recall that in the example output of `explicate-control` in Figure 4.10, `block57` through `block60` are trivial blocks, they do nothing but jump to another block. The first goal of this challenge assignment is to remove those blocks. Figure 4.13 repeats the result of `explicate-control` on the left and shows the result of bypassing the trivial blocks on the right. Let us focus on `block61`. The `then` branch jumps to `block57`, which in turn jumps to `block55`. The optimized code on the right of Figure 4.13 bypasses `block57`, with the `then` branch jumping directly to `block55`. The story is similar for the `else` branch, as well as for the two branches in `block62`. After the jumps in `block61` and `block62` have been optimized in this way, there are no longer any jumps to blocks `block57` through `block60`, so they can be removed.

The name of this pass is `optimize-jumps`. We recommend implementing this pass in two phases. The first phrase builds a hash table that maps labels to possibly improved labels. The second phase changes the target of each `goto` to use the improved label. If the label is for a trivial block, then the hash table should map the label to the first non-trivial block that can be reached from this label by jumping through trivial blocks. If the label is for a non-trivial block, then the hash table should map the label to itself; we do not want to change jumps to non-trivial blocks.

The first phase can be accomplished by constructing an empty hash table, call it `short-cut`, and then iterating over the control flow graph. Each time you encouter a block that is just a `goto`, then update the hash table, mapping the block's source to the target of the `goto`. Also, the hash table may already have mapped some labels to the block's source, to you must iterate through the hash table and update all of those so that they instead map to the target of the `goto`.

For the second phase, we recommend iterating through the *tail* of each block in the program, updating the target of every `goto` according to the mapping in `short-cut`.

Exercise 20. Implement the `optimize-jumps` pass as a transformation from C_1 to C_1, coming after the `explicate-control` pass. Check that `optimize-jumps` removes trivial blocks in a few example programs. Then check that your compiler still passes all of your tests.

There is another opportunity for optimizing jumps that is apparent in the example of Figure 4.11. The `start` block end with a jump to `block7953` and there are no other jumps to `block7953` in the rest of the program. In

```
block62:
    tmp54 = (read);
    if (eq? tmp54 2) then
       goto block59;
    else
       goto block60;
block61:
    tmp53 = (read);
    if (eq? tmp53 0) then
       goto block57;
    else
       goto block58;
block60:
    goto block56;
block59:
    goto block55;
block58:
    goto block56;
block57:
    goto block55;
block56:
    return (+ 700 77);
block55:
    return (+ 10 32);
start:
    tmp52 = (read);
    if (eq? tmp52 1) then
       goto block61;
    else
       goto block62;
```

⇒

```
block62:
    tmp54 = (read);
    if (eq? tmp54 2) then
       goto block55;
    else
       goto block56;
block61:
    tmp53 = (read);
    if (eq? tmp53 0) then
       goto block55;
    else
       goto block56;
block56:
    return (+ 700 77);
block55:
    return (+ 10 32);
start:
    tmp52 = (read);
    if (eq? tmp52 1) then
       goto block61;
    else
       goto block62;
```

Figure 4.13: Optimize jumps by removing trivial blocks.

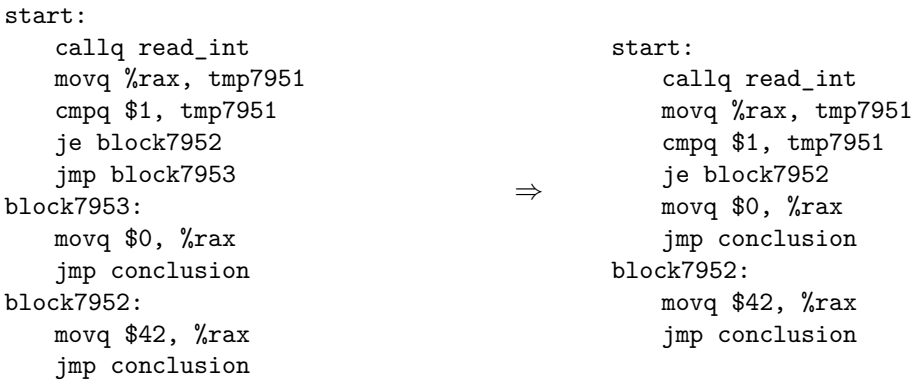


Figure 4.14: Merging basic blocks by removing unnecessary jumps.

this situation we can avoid the runtime overhead of this jump by merging `block7953` into the preceeding block, in this case the `start` block. Figure 4.14 shows the output of `select-instructions` on the left and the result of this optimization on the right.

Exercise 21. Implement a pass named `remove-jumps` that merges basic blocks into their preceeding basic block, when there is only one preceeding block. The pass should translate from psuedo $x86_1$ to pseudo $x86_1$ and it should come immediately after `select-instructions`. Check that `remove-jumps` accomplishes the goal of merging basic blocks on several test programs and check that your compiler passes all of your tests.

5

Tuples and Garbage Collection

In this chapter we study the implementation of mutable tuples (called "vectors" in Racket). This language feature is the first to use the computer's *heap* because the lifetime of a Racket tuple is indefinite, that is, a tuple lives forever from the programmer's viewpoint. Of course, from an implementer's viewpoint, it is important to reclaim the space associated with a tuple when it is no longer needed, which is why we also study *garbage collection garbage collection* techniques in this chapter.

Section 5.1 introduces the R_3 language including its interpreter and type checker. The R_3 language extends the R_2 language of Chapter 4 with vectors and Racket's `void` value. The reason for including the later is that the `vector-set!` operation returns a value of type `Void`[1].

Section 5.2 describes a garbage collection algorithm based on copying live objects back and forth between two halves of the heap. The garbage collector requires coordination with the compiler so that it can see all of the *root* pointers, that is, pointers in registers or on the procedure call stack.

Sections 5.4 through 5.10 discuss all the necessary changes and additions to the compiler passes, including a new compiler pass named `expose-allocation`.

[1]Racket's `Void` type corresponds to what is called the `Unit` type in the programming languages literature. Racket's `Void` type is inhabited by a single value `void` which corresponds to `unit` or `()` in the literature [Pierce, 2002].

type	::=	`Integer` | `Boolean` | (`Vector` *type* ...) | `Void`
exp	::=	*int* | (`read`) | (`-` *exp*) | (`+` *exp exp*) | (`-` *exp exp*)
	|	*var* | (`let` ([*var exp*]) *exp*)
	|	`#t` | `#f` | (`and` *exp exp*) | (`or` *exp exp*) | (`not` *exp*)
	|	(*cmp exp exp*) | (`if` *exp exp exp*)
	|	(`vector` *exp* ...) | (`vector-ref` *exp int*)
	|	(`vector-set!` *exp int exp*)
	|	(`void`) | (`has-type` *exp type*)
R_3	::=	*exp*

Figure 5.1: The concrete syntax of R_3, extending R_2 (Figure 4.1).

```
(let ([t (vector 40 #t (vector 2))])
  (if (vector-ref t 1)
      (+ (vector-ref t 0)
         (vector-ref (vector-ref t 2) 0))
      44))
```

Figure 5.2: Example program that creates tuples and reads from them.

5.1 The R_3 Language

Figure 5.1 defines the concrete syntax for R_3 and Figure 5.3 defines the abstract syntax. The R_3 language includes three new forms: `vector` for creating a tuple, `vector-ref` for reading an element of a tuple, and `vector-set!` for writing to an element of a tuple. The program in Figure 5.2 shows the usage of tuples in Racket. We create a 3-tuple `t` and a 1-tuple that is stored at index 2 of the 3-tuple, demonstrating that tuples are first-class values. The element at index 1 of `t` is `#t`, so the "then" branch of the `if` is taken. The element at index 0 of `t` is `40`, to which we add `2`, the element at index 0 of the 1-tuple. So the result of the program is `42`.

Tuples are our first encounter with heap-allocated data, which raises several interesting issues. First, variable binding performs a shallow-copy when dealing with tuples, which means that different variables can refer to the same tuple, that is, different variables can be *aliases* for the same entity. Consider the following example in which both `t1` and `t2` refer to the same tuple. Thus, the mutation through `t2` is visible when referencing the tuple from `t1`, so the result of this program is `42`.

```
exp  ::= (Int int) | (Prim 'read '()) | (Prim '- (list exp ))
       | (Prim '+ (list exp exp)) | (Prim '- (list exp exp))
       | (Var var) | (Let var exp exp)
       | (Bool bool) | (Prim 'and (list exp exp))
       | (Prim 'or (list exp exp)) | (Prim 'not (list exp ))
       | (Prim cmp (list exp exp)) | (If exp exp exp)
       | (Prim 'vector (list exp*))
       | (Prim 'vector-ref (list exp (Int int)))
       | (Prim 'vector-set! (list exp (Int int) exp))
       | (Void) | (HasType exp type)
R3   ::= (Program '() exp)
```

Figure 5.3: The abstract syntax of R_3.

```
(let ([t1 (vector 3 7)])
  (let ([t2 t1])
    (let ([_ (vector-set! t2 0 42)])
      (vector-ref t1 0))))
```

The next issue concerns the lifetime of tuples. Of course, they are created by the `vector` form, but when does their lifetime end? Notice that R_3 does not include an operation for deleting tuples. Furthermore, the lifetime of a tuple is not tied to any notion of static scoping. For example, the following program returns `42` even though the variable `w` goes out of scope prior to the `vector-ref` that reads from the vector it was bound to.

```
(let ([v (vector (vector 44))])
  (let ([x (let ([w (vector 42)])
             (let ([_ (vector-set! v 0 w)])
               0))])
    (+ x (vector-ref (vector-ref v 0) 0))))
```

From the perspective of programmer-observable behavior, tuples live forever. Of course, if they really lived forever, then many programs would run out of memory.[2] A Racket implementation must therefore perform automatic garbage collection.

[2]The R_3 language does not have looping or recursive functions, so it is nigh impossible to write a program in R_3 that will run out of memory. However, we add recursive functions in the next Chapter!

```
(define primitives (set ... 'vector 'vector-ref 'vector-set!))

(define (interp-op op)
  (match op
     ...
     ['vector vector]
     ['vector-ref vector-ref]
     ['vector-set! vector-set!]
     [else (error 'interp-op "unknown operator")]))

(define (interp-exp env)
  (lambda (e)
    (define recur (interp-exp env))
    (match e
       ...
      )))

(define (interp-R3 p)
 (match p
   [(Program '() e)
    ((interp-exp '()) e)]
  ))
```

Figure 5.4: Interpreter for the R_3 language.

Figure 5.4 shows the definitional interpreter for the R_3 language. We define the `vector`, `vector-ref`, and `vector-set!` operations for R_3 in terms of the corresponding operations in Racket. One subtle point is that the `vector-set!` operation returns the `#<void>` value. The `#<void>` value can be passed around just like other values inside an R_3 program and a `#<void>` value can be compared for equality with another `#<void>` value. However, there are no other operations specific to the the `#<void>` value in R_3. In contrast, Racket defines the `void?` predicate that returns `#t` when applied to `#<void>` and `#f` otherwise.

Figure 5.5 shows the type checker for R_3, which deserves some explanation. As we shall see in Section 5.2, we need to know which variables contain pointers into the heap, that is, which variables contain vectors. Also, when allocating a vector, we need to know which elements of the vector are pointers. We can obtain this information during type checking. The type checker in Figure 5.5 not only computes the type of an expression, it also wraps every sub-expression e with the form (`HasType` e T), where T is e's type.

Subsequently, in the `uncover-locals` pass (Section 5.7) this type information is propagated to all variables (including the temporaries generated by `remove-complex-opera*`).

To create the s-expression for the `Vector` type in Figure 5.5, we use the unquote-splicing operator `,@` to insert the list `t*` without its usual start and end parentheses.

5.2 Garbage Collection

Here we study a relatively simple algorithm for garbage collection that is the basis of state-of-the-art garbage collectors [Lieberman and Hewitt, 1983, Ungar, 1984, Jones and Lins, 1996, Detlefs et al., 2004, Dybvig, 2006, Tene et al., 2011]. In particular, we describe a two-space copying collector [Wilson, 1992] that uses Cheney's algorithm to perform the copy [Cheney, 1970]. Figure 5.6 gives a coarse-grained depiction of what happens in a two-space collector, showing two time steps, prior to garbage collection (on the top) and after garbage collection (on the bottom). In a two-space collector, the heap is divided into two parts named the FromSpace and the ToSpace. Initially, all allocations go to the FromSpace until there is not enough room for the next allocation request. At that point, the garbage collector goes to work to make more room.

The garbage collector must be careful not to reclaim tuples that will be used by the program in the future. Of course, it is impossible in general to predict what a program will do, but we can over approximate the will-be-used tuples by preserving all tuples that could be accessed by *any* program given the current computer state. A program could access any tuple whose address is in a register or on the procedure call stack. These addresses are called the *root set*. In addition, a program could access any tuple that is transitively reachable from the root set. Thus, it is safe for the garbage collector to reclaim the tuples that are not reachable in this way.

So the goal of the garbage collector is twofold:

1. preserve all tuple that are reachable from the root set via a path of pointers, that is, the *live* tuples, and

2. reclaim the memory of everything else, that is, the *garbage*.

A copying collector accomplishes this by copying all of the live objects from the FromSpace into the ToSpace and then performs a slight of hand, treating the ToSpace as the new FromSpace and the old FromSpace as the new ToSpace. In the example of Figure 5.6, there are three pointers in the root

```
(define (type-check-exp env)
  (lambda (e)
    (define recur (type-check-exp env))
    (match e
      ...
      [(Void) (values (HasType (Void) 'Void) 'Void)]
      [(Prim 'vector es)
       (define-values (e* t*) (for/lists (e* t*) ([e es]) (recur e)))
       (let ([t `(Vector ,@t*)])
         (values (HasType (Prim 'vector e*) t) t))]
      [(Prim 'vector-ref (list e (Int i)))
       (define-values (e^ t) (recur e))
       (match t
         [`(Vector ,ts ...)
          (unless (and (exact-nonnegative-integer? i) (< i (length ts)))
            (error 'type-check-exp "invalid index ~a" i))
          (let ([t (list-ref ts i)])
            (values
               (HasType (Prim 'vector-ref
                              (list e^ (HasType (Int i) 'Integer)))
                  t)
               t))]
         [else (error "expected a vector in vector-ref, not" t)])]
      [(Prim 'eq? (list e1 e2))
       (define-values (e1^ T1) (recur e1))
       (define-values (e2^ T2) (recur e2))
       (unless (equal? T1 T2)
         (error "arguments of eq? must have the same type, but are not"
                (list T1 T2)))
       (values (HasType (Prim 'eq? (list e1^ e2^)) 'Boolean) 'Boolean)]
      ...
      )))
```

Figure 5.5: Type checker for the R_3 language.

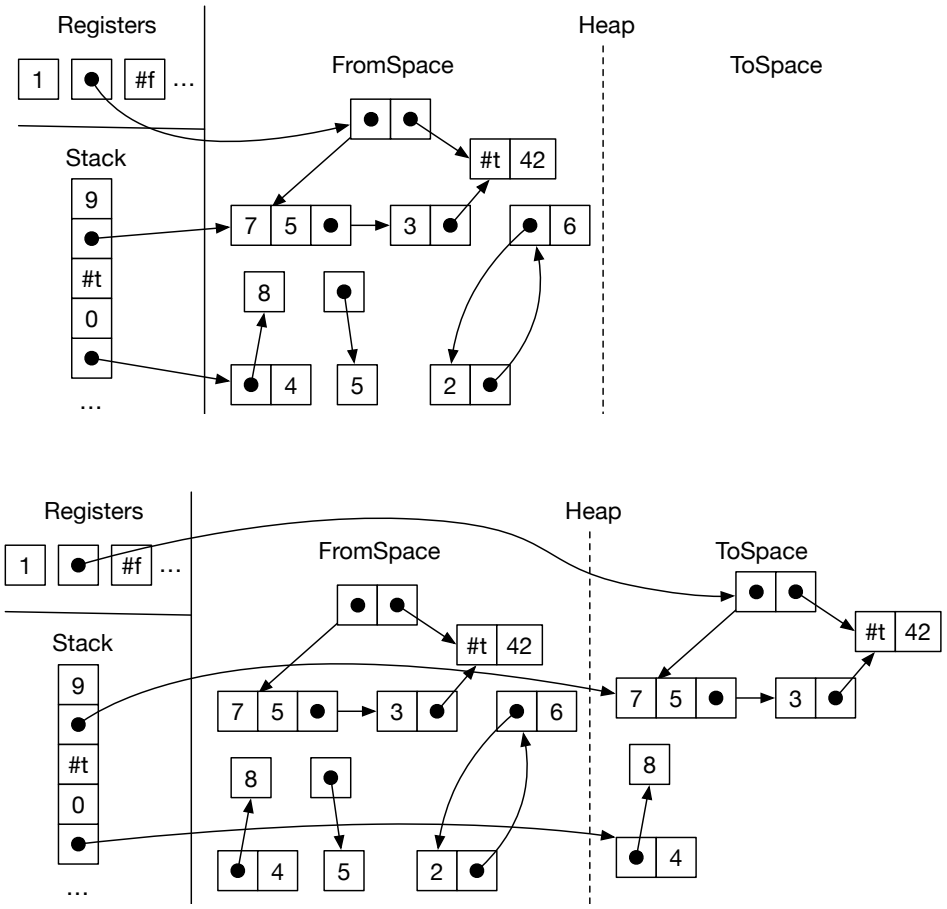


Figure 5.6: A copying collector in action.

set, one in a register and two on the stack. All of the live objects have been copied to the ToSpace (the right-hand side of Figure 5.6) in a way that preserves the pointer relationships. For example, the pointer in the register still points to a 2-tuple whose first element is a 3-tuple and whose second element is a 2-tuple. There are four tuples that are not reachable from the root set and therefore do not get copied into the ToSpace.

The exact situation in Figure 5.6 cannot be created by a well-typed program in R_3 because it contains a cycle. However, creating cycles will be possible once we get to R_6. We design the garbage collector to deal with cycles to begin with so we will not need to revisit this issue.

There are many alternatives to copying collectors (and their bigger sib-

lings, the generational collectors) when its comes to garbage collection, such as mark-and-sweep [McCarthy, 1960] and reference counting [Collins, 1960]. The strengths of copying collectors are that allocation is fast (just a comparison and pointer increment), there is no fragmentation, cyclic garbage is collected, and the time complexity of collection only depends on the amount of live data, and not on the amount of garbage [Wilson, 1992]. The main disadvantages of a two-space copying collector is that it uses a lot of space and takes a long time to perform the copy, though these problems are ameliorated in generational collectors. Racket and Scheme programs tend to allocate many small objects and generate a lot of garbage, so copying and generational collectors are a good fit. Garbage collection is an active research topic, especially concurrent garbage collection [Tene et al., 2011]. Researchers are continuously developing new techniques and revisiting old trade-offs [Blackburn et al., 2004, Jones et al., 2011, Shahriyar et al., 2013, Cutler and Morris, 2015, Shidal et al., 2015, Österlund and Löwe, 2016, Jacek and Moss, 2019, Gamari and Dietz, 2020]. Researchers meet every year at the International Symposium on Memory Management to present these findings.

5.2.1 Graph Copying via Cheney's Algorithm

Let us take a closer look at the copying of the live objects. The allocated objects and pointers can be viewed as a graph and we need to copy the part of the graph that is reachable from the root set. To make sure we copy all of the reachable vertices in the graph, we need an exhaustive graph traversal algorithm, such as depth-first search or breadth-first search [Moore, 1959, Cormen et al., 2001]. Recall that such algorithms take into account the possibility of cycles by marking which vertices have already been visited, so as to ensure termination of the algorithm. These search algorithms also use a data structure such as a stack or queue as a to-do list to keep track of the vertices that need to be visited. We shall use breadth-first search and a trick due to Cheney [1970] for simultaneously representing the queue and copying tuples into the ToSpace.

Figure 5.7 shows several snapshots of the ToSpace as the copy progresses. The queue is represented by a chunk of contiguous memory at the beginning of the ToSpace, using two pointers to track the front and the back of the queue. The algorithm starts by copying all tuples that are immediately reachable from the root set into the ToSpace to form the initial queue. When we copy a tuple, we mark the old tuple to indicate that it has been visited. We discuss how this marking is accomplish in Section 5.2.2. Note

that any pointers inside the copied tuples in the queue still point back to the FromSpace. Once the initial queue has been created, the algorithm enters a loop in which it repeatedly processes the tuple at the front of the queue and pops it off the queue. To process a tuple, the algorithm copies all the tuple that are directly reachable from it to the ToSpace, placing them at the back of the queue. The algorithm then updates the pointers in the popped tuple so they point to the newly copied tuples.

Getting back to Figure 5.7, in the first step we copy the tuple whose second element is 42 to the back of the queue. The other pointer goes to a tuple that has already been copied, so we do not need to copy it again, but we do need to update the pointer to the new location. This can be accomplished by storing a *forwarding pointer* to the new location in the old tuple, back when we initially copied the tuple into the ToSpace. This completes one step of the algorithm. The algorithm continues in this way until the front of the queue is empty, that is, until the front catches up with the back.

5.2.2 Data Representation

The garbage collector places some requirements on the data representations used by our compiler. First, the garbage collector needs to distinguish between pointers and other kinds of data. There are several ways to accomplish this.

1. Attached a tag to each object that identifies what type of object it is [McCarthy, 1960].

2. Store different types of objects in different regions [Steele, 1977].

3. Use type information from the program to either generate type-specific code for collecting or to generate tables that can guide the collector [Appel, 1989, Goldberg, 1991, Diwan et al., 1992].

Dynamically typed languages, such as Lisp, need to tag objects anyways, so option 1 is a natural choice for those languages. However, R_3 is a statically typed language, so it would be unfortunate to require tags on every object, especially small and pervasive objects like integers and Booleans. Option 3 is the best-performing choice for statically typed languages, but comes with a relatively high implementation complexity. To keep this chapter within a 2-week time budget, we recommend a combination of options 1 and 2, using separate strategies for the stack and the heap.

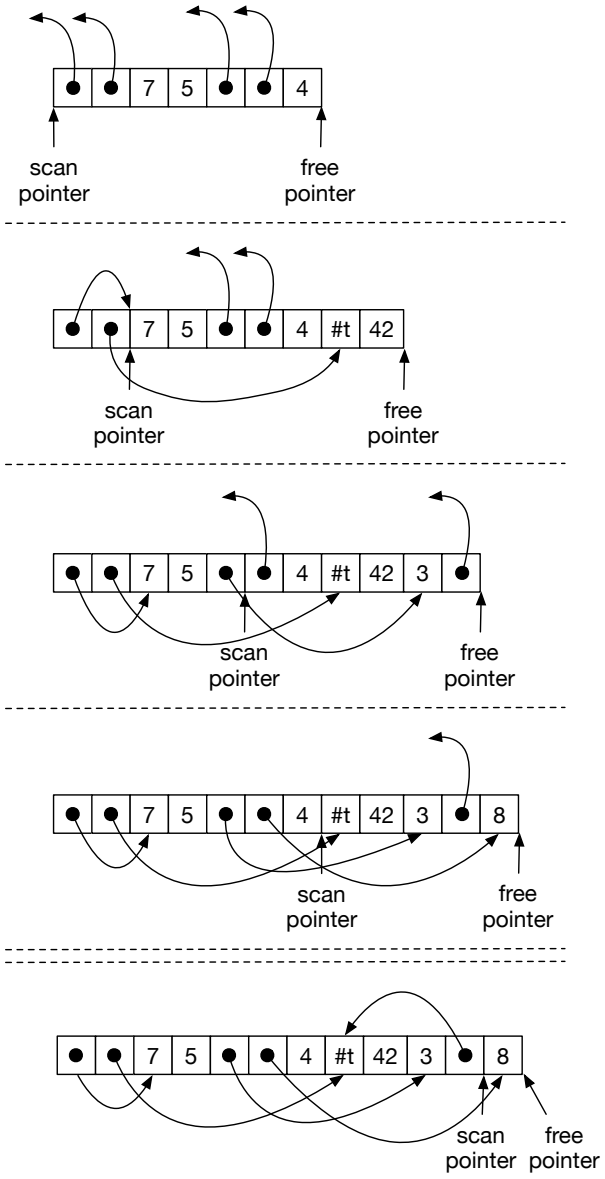


Figure 5.7: Depiction of the Cheney algorithm copying the live tuples.

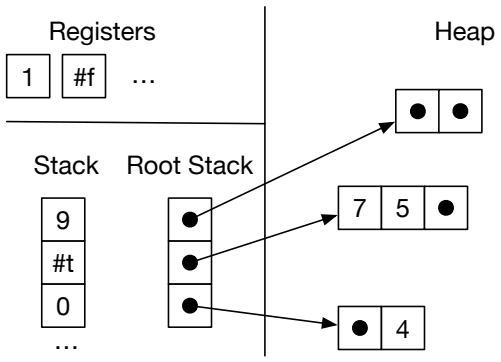


Figure 5.8: Maintaining a root stack to facilitate garbage collection.

Regarding the stack, we recommend using a separate stack for pointers, which we call a *root stack* (a.k.a. "shadow stack") [Siebert, 2001, Henderson, 2002, Baker et al., 2009]. That is, when a local variable needs to be spilled and is of type `(Vector` $type_1 \ldots type_n$`)`, then we put it on the root stack instead of the normal procedure call stack. Furthermore, we always spill vector-typed variables if they are live during a call to the collector, thereby ensuring that no pointers are in registers during a collection. Figure 5.8 reproduces the example from Figure 5.6 and contrasts it with the data layout using a root stack. The root stack contains the two pointers from the regular stack and also the pointer in the second register.

The problem of distinguishing between pointers and other kinds of data also arises inside of each tuple on the heap. We solve this problem by attaching a tag, an extra 64-bits, to each tuple. Figure 5.9 zooms in on the tags for two of the tuples in the example from Figure 5.6. Note that we have drawn the bits in a big-endian way, from right-to-left, with bit location 0 (the least significant bit) on the far right, which corresponds to the direction of the x86 shifting instructions `salq` (shift left) and `sarq` (shift right). Part of each tag is dedicated to specifying which elements of the tuple are pointers, the part labeled "pointer mask". Within the pointer mask, a 1 bit indicates there is a pointer and a 0 bit indicates some other kind of data. The pointer mask starts at bit location 7. We have limited tuples to a maximum size of 50 elements, so we just need 50 bits for the pointer mask. The tag also contains two other pieces of information. The length of the tuple (number of elements) is stored in bits location 1 through 6. Finally, the bit at location 0 indicates whether the tuple has yet to be copied to the ToSpace. If the

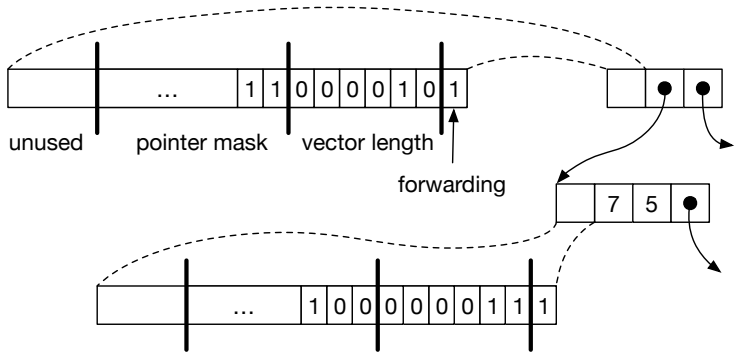


Figure 5.9: Representation of tuples in the heap.

bit has value 1, then this tuple has not yet been copied. If the bit has value 0 then the entire tag is a forwarding pointer. (The lower 3 bits of a pointer are always zero anyways because our tuples are 8-byte aligned.)

5.2.3 Implementation of the Garbage Collector

An implementation of the copying collector is provided in the `runtime.c` file. Figure 5.10 defines the interface to the garbage collector that is used by the compiler. The `initialize` function creates the FromSpace, ToSpace, and root stack and should be called in the prelude of the `main` function. The `initialize` function puts the address of the beginning of the FromSpace into the global variable `free_ptr`. The global variable `fromspace_end` points to the address that is 1-past the last element of the FromSpace. (We use half-open intervals to represent chunks of memory [Dijkstra, 1982].) The `rootstack_begin` variable points to the first element of the root stack.

As long as there is room left in the FromSpace, your generated code can allocate tuples simply by moving the `free_ptr` forward. The amount of room left in FromSpace is the difference between the `fromspace_end` and the `free_ptr`. The `collect` function should be called when there is not enough room left in the FromSpace for the next allocation. The `collect` function takes a pointer to the current top of the root stack (one past the last item that was pushed) and the number of bytes that need to be allocated. The `collect` function performs the copying collection and leaves the heap in a state such that the next allocation will succeed.

The introduction of garbage collection has a non-trivial impact on our compiler passes. We introduce two new compiler passes named `expose-allocation` and `uncover-locals`. We make significant changes to `select-instructions`,

```
void initialize(uint64_t rootstack_size, uint64_t heap_size);
void collect(int64_t** rootstack_ptr, uint64_t bytes_requested);
int64_t* free_ptr;
int64_t* fromspace_begin;
int64_t* fromspace_end;
int64_t** rootstack_begin;
```

Figure 5.10: The compiler's interface to the garbage collector.

`build-interference`, `allocate-registers`, and `print-x86` and make minor changes in severl more passes. The following program will serve as our running example. It creates two tuples, one nested inside the other. Both tuples have length one. The program accesses the element in the inner tuple tuple via two vector references.

```
(vector-ref (vector-ref (vector (vector 42)) 0) 0))
```

5.3 Shrink

Recall that the `shrink` pass translates the primitives operators into a smaller set of primitives. Because this pass comes after type checking, but before the passes that require the type information in the `HasType` AST nodes, the `shrink` pass must be modified to wrap `HasType` around each AST node that it generates.

5.4 Expose Allocation

The pass `expose-allocation` lowers the `vector` creation form into a conditional call to the collector followed by the allocation. We choose to place the `expose-allocation` pass before `remove-complex-opera*` because the code generated by `expose-allocation` contains complex operands. We also place `expose-allocation` before `explicate-control` because `expose-allocation` introduces new variables using `let`, but `let` is gone after `explicate-control`.

The output of `expose-allocation` is a language R_3' that extends R_3 with the three new forms that we use in the translation of the `vector` form.

$$\mathit{exp} \quad ::= \quad \cdots \mid (\texttt{collect}\ \mathit{int}) \mid (\texttt{allocate}\ \mathit{int}\ \mathit{type}) \mid (\texttt{global-value}\ \mathit{name})$$

The (`collect` n) form runs the garbage collector, requesting n bytes. It will become a call to the `collect` function in `runtime.c` in `select-instructions`.

The (`allocate` n T) form creates an tuple of n elements. The T parameter is the type of the tuple: (`Vector` $type_1 \ldots type_n$) where $type_i$ is the type of the ith element in the tuple. The (`global-value` *name*) form reads the value of a global variable, such as `free_ptr`.

In the following, we show the transformation for the `vector` form into 1) a sequence of let-bindings for the initializing expressions, 2) a conditional call to `collect`, 3) a call to `allocate`, and 4) the initialization of the vector. In the following, *len* refers to the length of the vector and *bytes* is how many total bytes need to be allocated for the vector, which is 8 for the tag plus *len* times 8.

```
  (has-type (vector e_0 ... e_{n-1}) type)
⟹
  (let ([x_0 e_0]) ... (let ([x_{n-1} e_{n-1}])
  (let ([_ (if (< (+ (global-value free_ptr) bytes)
                  (global-value fromspace_end))
               (void)
               (collect bytes))])
  (let ([v (allocate len type)])
  (let ([_ (vector-set! v 0 x_0)]) ...
  (let ([_ (vector-set! v n-1 x_{n-1})])
    v) ... )))) ...)
```

In the above, we suppressed all of the `has-type` forms in the output for the sake of readability. The placement of the initializing expressions $e_0, \ldots, e_{n-1}$ prior to the `allocate` and the sequence of `vector-set!` is important, as those expressions may trigger garbage collection and we cannot have an allocated but uninitialized tuple on the heap during a collection.

Figure 5.11 shows the output of the `expose-allocation` pass on our running example.

5.5 Remove Complex Operands

The new forms `collect`, `allocate`, and `global-value` should all be treated as complex operands. A new case for `HasType` is needed and the case for `Prim` needs to be handled carefully to prevent the `Prim` node from being separated from its enclosing `HasType`.

5.6 Explicate Control and the C_2 language

The output of `explicate-control` is a program in the intermediate language C_2, whose concrete syntax is defined in Figure 5.12 and whose ab-

```
(vector-ref
 (vector-ref
  (let ([vecinit7976
         (let ([vecinit7972 42])
           (let ([collectret7974
                  (if (< (+ (global-value free_ptr) 16)
                         (global-value fromspace_end))
                      (void)
                      (collect 16)
                      )])
             (let ([alloc7971 (allocate 1 (Vector Integer))])
               (let ([initret7973 (vector-set! alloc7971 0 vecinit7972)])
                 alloc7971)
               )
             )
           )
         ])
    (let ([collectret7978
           (if (< (+ (global-value free_ptr) 16)
                  (global-value fromspace_end))
               (void)
               (collect 16)
               )])
      (let ([alloc7975 (allocate 1 (Vector (Vector Integer)))])
        (let ([initret7977 (vector-set! alloc7975 0 vecinit7976)])
          alloc7975)
        )
      )
    )
  0)
 0)
```

Figure 5.11: Output of the `expose-allocation` pass, minus all of the `has-type` forms.

```
atm    ::=  int | var | bool
cmp    ::=  eq? | <
exp    ::=  atm | (read) | (- atm) | (+ atm atm)
          |  (not atm) | (cmp atm atm)
          |  (allocate int type)
          |  (vector-ref atm int) | (vector-set! atm int atm)
          |  (global-value var) | (void)
stmt   ::=  var = exp; | (collect int)
tail   ::=  return exp; | stmt tail | goto label;
          |  if (cmp atm atm) goto label; else goto label;
C_2    ::=  (label: tail) ...
```

Figure 5.12: The concrete syntax of the C_2 intermediate language.

```
atm    ::=  (Int int) | (Var var) | (Bool bool)
cmp    ::=  eq? | <
exp    ::=  atm | (Prim 'read '())
          |  (Prim '- (list atm )) | (Prim '+ (list atm atm))
          |  (Prim not (list atm)) | (Prim cmp (list atm atm))
          |  (Allocate int type)
          |  (Prim 'vector-ref (list atm (Int int)))
          |  (Prim 'vector-set! (list atm (Int int) atm))
          |  (GlobalValue var) | (Void)
stmt   ::=  (Assign (Var var) exp) | (Collect int)
tail   ::=  (Return exp) | (Seq stmt tail) | (Goto label)
          |  (IfStmt (Prim cmp (list atm atm)) (Goto label) (Goto label))
C_2    ::=  (Program info (CFG (label . tail) ...))
```

Figure 5.13: The abstract syntax of C_2, extending C_1 (Figure 4.8).

stract syntax is defined in Figure 5.13. The new forms of C_2 include the `allocate`, `vector-ref`, and `vector-set!`, and `global-value` expressions and the `collect` statement. The `explicate-control` pass can treat these new forms much like the other forms.

5.7 Uncover Locals

Recall that the `explicate-control` function collects all of the local variables so that it can store them in the *info* field of the `Program` structure. Also recall that we need to know the types of all the local variables for purposes of identifying the root set for the garbage collector. Thus, we create a pass named `uncover-locals` to collect not just the variables but the variables and their types in the form of an alist. Thanks to the `HasType` nodes, the types are readily available at every assignment to a variable. We recommend storing the resulting alist in the *info* field of the program, associated with the `locals` key. Figure 5.14 lists the output of the `uncover-locals` pass on the running example.

```
locals:
    vecinit7976 : '(Vector Integer), tmp7980 : 'Integer,
    alloc7975 : '(Vector (Vector Integer)), tmp7983 : 'Integer,
    collectret7974 : 'Void, initret7977 : 'Void,
    collectret7978 : 'Void, tmp7985 : '(Vector Integer),
    tmp7984 : 'Integer, tmp7979 : 'Integer, tmp7982 : 'Integer,
    alloc7971 : '(Vector Integer), tmp7981 : 'Integer,
    vecinit7972 : 'Integer, initret7973 : 'Void,
block91:
    (collect 16)
    goto block89;
block90:
    collectret7974 = (void);
    goto block89;
block89:
    alloc7971 = (allocate 1 (Vector Integer));
    initret7973 = (vector-set! alloc7971 0 vecinit7972);
    vecinit7976 = alloc7971;
    tmp7982 = (global-value free_ptr);
    tmp7983 = (+ tmp7982 16);
    tmp7984 = (global-value fromspace_end);
    if (< tmp7983 tmp7984) then
       goto block87;
    else
       goto block88;
block88:
    (collect 16)
    goto block86;
block87:
    collectret7978 = (void);
    goto block86;
block86:
    alloc7975 = (allocate 1 (Vector (Vector Integer)));
    initret7977 = (vector-set! alloc7975 0 vecinit7976);
    tmp7985 = (vector-ref alloc7975 0);
    return (vector-ref tmp7985 0);
start:
    vecinit7972 = 42;
    tmp7979 = (global-value free_ptr);
    tmp7980 = (+ tmp7979 16);
    tmp7981 = (global-value fromspace_end);
    if (< tmp7980 tmp7981) then
       goto block90;
    else
       goto block91;
```

Figure 5.14: Output of `uncover-locals` for the running example.

5.8 Select Instructions and the $\mathbf{x86}_2$ Language

In this pass we generate x86 code for most of the new operations that were needed to compile tuples, including `Allocate`, `Collect`, `vector-ref`, `vector-set!`, and `void`. We compile `GlobalValue` to `Global` because the later has a different concrete syntax (see Figures 5.15 and 5.16).

The `vector-ref` and `vector-set!` forms translate into `movq` instructions. (The plus one in the offset is to get past the tag at the beginning of the tuple representation.)

lhs `= (vector-ref` *vec n*`);`
$\Longrightarrow$
`movq` *vec*′`, %r11`
`movq` $8(n+1)$`(%r11),` *lhs*′

lhs `= (vector-set!` *vec n arg*`);`
$\Longrightarrow$
`movq` *vec*′`, %r11`
`movq` *arg*′`,` $8(n+1)$`(%r11)`
`movq $0,` *lhs*′

The *lhs*′, *vec*′, and *arg*′ are obtained by translating *vec* and *arg* to x86. The move of *vec*′ to register `r11` ensures that offset expression $-8(n+1)$`(%r11)` contains a register operand. This requires removing `r11` from consideration by the register allocating.

Why not use `rax` instead of `r11`? Suppose we instead used `rax`. Then the generated code for `vector-set!` would be

`movq` *vec*′`, %rax`
`movq` *arg*′`,` $8(n+1)$`(%rax)`
`movq $0,` *lhs*′

Next, suppose that *arg*′ ends up as a stack location, so `patch-instructions` would insert a move through `rax` as follows.

`movq` *vec*′`, %rax`
`movq` *arg*′`, %rax`
`movq %rax,` $8(n+1)$`(%rax)`
`movq $0,` *lhs*′

But the above sequence of instructions does not work because we're trying to use `rax` for two different values (*vec*′ and *arg*′) at the same time!

We compile the `allocate` form to operations on the `free_ptr`, as shown below. The address in the `free_ptr` is the next free address in the FromSpace, so we move it into the *lhs* and then move it forward by enough space for

arg	::=	$*int* | %*reg* | *int*(%*reg*) | %*bytereg* | *var*(%rip)
$x86_1$	::=	.globl main main: *instr* ...

Figure 5.15: The concrete syntax of $x86_2$ (extends $x86_1$ of Figure 4.5).

arg	::=	(Int *int*) | (Reg *reg*) | (Deref *reg int*) | (ByteReg *reg*)
	|	(Global *var*)
$x86_2$	::=	(Program *info* (CFG ((*label* . *block*) ...))

Figure 5.16: The abstract syntax of $x86_2$ (extends $x86_1$ of Figure 4.6).

the tuple being allocated, which is $8(len + 1)$ bytes because each element is 8 bytes (64 bits) and we use 8 bytes for the tag. Last but not least, we initialize the *tag*. Refer to Figure 5.9 to see how the tag is organized. We recommend using the Racket operations `bitwise-ior` and `arithmetic-shift` to compute the tag during compilation. The type annotation in the `vector` form is used to determine the pointer mask region of the tag.

lhs = (allocate *len* (Vector *type* ...));
$\Longrightarrow$
movq free_ptr(%rip), lhs'
addq $8(len + 1)$, free_ptr(%rip)
movq lhs', %r11
movq $*tag*, 0(%r11)

The `collect` form is compiled to a call to the `collect` function in the runtime. The arguments to `collect` are 1) the top of the root stack and 2) the number of bytes that need to be allocated. We shall use another dedicated register, `r15`, to store the pointer to the top of the root stack. So `r15` is not available for use by the register allocator.

(collect *bytes*)
$\Longrightarrow$
movq %r15, %rdi
movq $*bytes*, %rsi
callq collect

The concrete and abstract syntax of the $x86_2$ language is defined in Figures 5.15 and 5.16. It differs from $x86_1$ just in the addition of the form for global variables. Figure 5.17 shows the output of the `select-instructions` pass on the running example.

```
block35:
    movq free_ptr(%rip), alloc9024
    addq $16, free_ptr(%rip)
    movq alloc9024, %r11
    movq $131, 0(%r11)
    movq alloc9024, %r11
    movq vecinit9025, 8(%r11)
    movq $0, initret9026
    movq alloc9024, %r11
    movq 8(%r11), tmp9034
    movq tmp9034, %r11
    movq 8(%r11), %rax
    jmp conclusion
block36:
    movq $0, collectret9027
    jmp block35
block38:
    movq free_ptr(%rip), alloc9020
    addq $16, free_ptr(%rip)
    movq alloc9020, %r11
    movq $3, 0(%r11)
    movq alloc9020, %r11
    movq vecinit9021, 8(%r11)
    movq $0, initret9022
    movq alloc9020, vecinit9025
    movq free_ptr(%rip), tmp9031
    movq tmp9031, tmp9032
    addq $16, tmp9032
    movq fromspace_end(%rip), tmp9033
    cmpq tmp9033, tmp9032
    jl block36
    jmp block37
block37:
    movq %r15, %rdi
    movq $16, %rsi
    callq 'collect
    jmp block35
block39:
    movq $0, collectret9023
    jmp block38
```

```
start:
    movq $42, vecinit9021
    movq free_ptr(%rip), tmp9028
    movq tmp9028, tmp9029
    addq $16, tmp9029
    movq fromspace_end(%rip), tmp9030
    cmpq tmp9030, tmp9029
    jl block39
    jmp block40
block40:
    movq %r15, %rdi
    movq $16, %rsi
    callq 'collect
    jmp block38
```

Figure 5.17: Output of the `select-instructions` pass.

5.9 Register Allocation

As discussed earlier in this chapter, the garbage collector needs to access all the pointers in the root set, that is, all variables that are vectors. It will be the responsibility of the register allocator to make sure that:

1. the root stack is used for spilling vector-typed variables, and
2. if a vector-typed variable is live during a call to the collector, it must be spilled to ensure it is visible to the collector.

The later responsibility can be handled during construction of the inference graph, by adding interference edges between the call-live vector-typed variables and all the callee-saved registers. (They already interfere with the caller-saved registers.) The type information for variables is in the `Program` form, so we recommend adding another parameter to the `build-interference` function to communicate this alist.

The spilling of vector-typed variables to the root stack can be handled after graph coloring, when choosing how to assign the colors (integers) to registers and stack locations. The `Program` output of this pass changes to also record the number of spills to the root stack.

5.10 Print x86

Figure 5.18 shows the output of the `print-x86` pass on the running example. In the prelude and conclusion of the `main` function, we treat the root stack very much like the regular stack in that we move the root stack pointer (`r15`) to make room for the spills to the root stack, except that the root stack grows up instead of down. For the running example, there was just one spill so we increment `r15` by 8 bytes. In the conclusion we decrement `r15` by 8 bytes.

One issue that deserves special care is that there may be a call to `collect` prior to the initializing assignments for all the variables in the root stack. We do not want the garbage collector to accidentally think that some uninitialized variable is a pointer that needs to be followed. Thus, we zero-out all locations on the root stack in the prelude of `main`. In Figure 5.18, the instruction `movq $0, (%r15)` accomplishes this task. The garbage collector tests each root to see if it is null prior to dereferencing it.

Figure 5.19 gives an overview of all the passes needed for the compilation of R_3.

```
block35:
       movq   free_ptr(%rip), %rcx
       addq   $16, free_ptr(%rip)
       movq   %rcx, %r11
       movq   $131, 0(%r11)
       movq   %rcx, %r11
       movq   -8(%r15), %rax
       movq   %rax, 8(%r11)
       movq   $0, %rdx
       movq   %rcx, %r11
       movq   8(%r11), %rcx
       movq   %rcx, %r11
       movq   8(%r11), %rax
       jmp conclusion
block36:
       movq   $0, %rcx
       jmp block35
block38:
       movq   free_ptr(%rip), %rcx
       addq   $16, free_ptr(%rip)
       movq   %rcx, %r11
       movq   $3, 0(%r11)
       movq   %rcx, %r11
       movq   %rbx, 8(%r11)
       movq   $0, %rdx
       movq   %rcx, -8(%r15)
       movq   free_ptr(%rip), %rcx
       addq   $16, %rcx
       movq   fromspace_end(%rip), %rdx
       cmpq   %rdx, %rcx
       jl block36
       movq   %r15, %rdi
       movq   $16, %rsi
       callq  collect
       jmp block35
block39:
       movq   $0, %rcx
       jmp block38
```

```
start:
       movq   $42, %rbx
       movq   free_ptr(%rip), %rdx
       addq   $16, %rdx
       movq   fromspace_end(%rip), %rcx
       cmpq   %rcx, %rdx
       jl block39
       movq   %r15, %rdi
       movq   $16, %rsi
       callq  collect
       jmp block38

       .globl main
main:
       pushq  %rbp
       movq   %rsp, %rbp
       pushq  %r13
       pushq  %r12
       pushq  %rbx
       pushq  %r14
       subq   $0, %rsp
       movq $16384, %rdi
       movq $16, %rsi
       callq initialize
       movq rootstack_begin(%rip), %r15
       movq $0, (%r15)
       addq $8, %r15
       jmp start
conclusion:
       subq $8, %r15
       addq   $0, %rsp
       popq   %r14
       popq   %rbx
       popq   %r12
       popq   %r13
       popq   %rbp
       retq
```

Figure 5.18: Output of the `print-x86` pass.

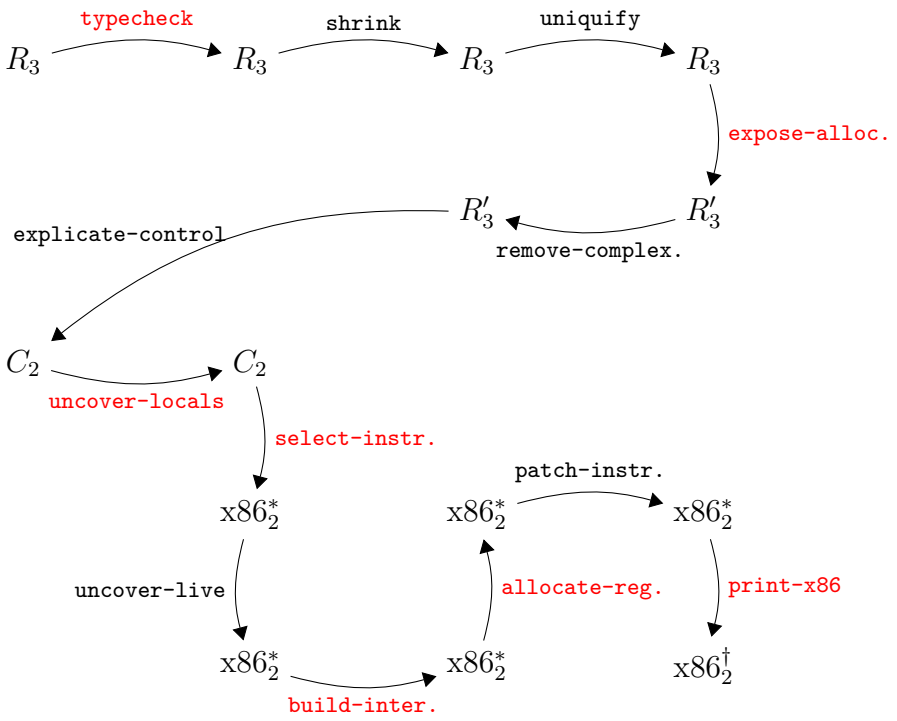


Figure 5.19: Diagram of the passes for R_3, a language with tuples.

type	::=	`Integer` | `Boolean` | (`Vector` *type* ...) | `Void` | *var*
cmp	::=	`eq?` | `<` | `<=` | `>` | `>=`
exp	::=	*int* | (`read`) | (`-` *exp*) | (`+` *exp exp*) | (`-` *exp exp*)
	|	*var* | (`let` ([*var exp*]) *exp*)
	|	`#t` | `#f` | (`and` *exp exp*) | (`or` *exp exp*) | (`not` *exp*)
	|	(*cmp exp exp*) | (`if` *exp exp exp*)
	|	(`vector` *exp* ...) | (`vector-ref` *exp int*)
	|	(`vector-set!` *exp int exp*)
	|	(`void`) | (*var exp* ...)
def	::=	(`struct` *var* ([*var* : *type*] ...) `#:mutable`)
R_3	::=	*def* ... *exp*

Figure 5.20: The concrete syntax of R_3^s, extending R_3 (Figure 5.1).

5.11 Challenge: Simple Structures

Figure 5.20 defines the concrete syntax for R_3^s, which extends R^3 with support for simple structures. Recall that a `struct` in Typed Racket is a user-defined data type that contains named fields and that is heap allocated, similar to a vector. The following is an example of a structure definition, in this case the definition of a `point` type.

```
(struct point ([x : Integer] [y : Integer]) #:mutable)
```

An instance of a structure is created using function call syntax, with the name of the structure in the function position:

```
(point 7 12)
```

Function-call syntax is also used to read the value in a field of a structure. The function name is formed by the structure name, a dash, and the field name. The following example uses `point-x` and `point-y` to access the `x` and `y` fields of two point instances.

```
(let ([pt1 (point 7 12)])
  (let ([pt2 (point 4 3)])
    (+ (- (point-x pt1) (point-x pt2))
       (- (point-y pt1) (point-y pt2)))))
```

Similarly, to write to a field of a structure, use its set function, whose name starts with `set-`, followed by the structure name, then a dash, then the field name, and conclused with an exclamation mark. The folowing example uses `set-point-x!` to change the `x` field from `7` to `42`.

```
(let ([pt (point 7 12)])
  (let ([_ (set-point-x! pt 42)])
    (point-x pt)))
```

Exercise 22. Extend your compiler with support for simple structures, compiling R_3^s to x86 assembly code. Create five new test cases that use structures and test your compiler.

5.12 Challenge: Generational Collection

The copying collector described in Section 5.2 can incur significant runtime overhead because the call to `collect` takes time proportional to all of the live data. One way to reduce this overhead is to reduce how much data is inspected in each call to `collect`. In particular, researchers have observed that recently allocated data is more likely to become garbage then data that has survived one or more previous calls to `collect`. This insight motivated the creation of *generational garbage collectors* that 1) segragates data according to its age into two or more generations, 2) allocates less space for younger generations, so collecting them is faster, and more space for the older generations, and 3) performs collection on the younger generations more frequently then for older generations [Wilson, 1992].

For this challenge assignment, the goal is to adapt the copying collector implemented in `runtime.c` to use two generations, one for young data and one for old data. Each generation consists of a FromSpace and a ToSpace. The following is a sketch of how to adapt the `collect` function to use the two generations.

1. Copy the young generation's FromSpace to its ToSpace then switch the role of the ToSpace and FromSpace

2. If there is enough space for the requested number of bytes in the young FromSpace, then return from `collect`.

3. If there is not enough space in the young FromSpace for the requested bytes, then move the data from the young generation to the old one with the following steps:

   (a) If there is enough room in the old FromSpace, copy the young FromSpace to the old FromSpace and then return.

(b) If there is not enough room in the old FromSpace, then collect the old generation by copying the old FromSpace to the old ToSpace and swap the roles of the old FromSpace and ToSpace.

(c) If there is enough room now, copy the young FromSpace to the old FromSpace and return. Otherwise, allocate a larger FromSpace and ToSpace for the old generation. Copy the young FromSpace and the old FromSpace into the larger FromSpace for the old generation and then return.

We recommend that you generalize the `cheney` function so that it can be used for all the copies mentioned above: between the young FromSpace and ToSpace, between the old FromSpace and ToSpace, and between the young FromSpace and old FromSpace. This can be accomplished by adding parameters to `cheney` that replace its use of the global variables `fromspace_begin`, `fromspace_end`, `tospace_begin`, and `tospace_end`.

Note that the collection of the young generation does not traverse the old generation. This introduces a potential problem: there may be young data that is only reachable through pointers in the old generation. If these pointers are not taken into account, the collector could throw away young data that is live! One solution, called *pointer recording*, is to maintain a set of all the pointers from the old generation into the new generation and consider this set as part of the root set. To maintain this set, the compiler must insert extra instructions around every `vector-set!`. If the vector being modified is in the old generation, and if the value being written is a pointer into the new generation, than that pointer must be added to the set. Also, if the value being overwritten was a pointer into the new generation, then that pointer should be removed from the set.

Exercise 23. Adapt the `collect` function in `runtime.c` to implement generational garbage collection, as outlined in this section. Update the code generation for `vector-set!` to implement pointer recording. Make sure that your new compiler and runtime passes your test suite.

6

Functions

This chapter studies the compilation of functions similar to those found in the C language. This corresponds to a subset of Typed Racket in which only top-level function definitions are allowed. This kind of function is an important stepping stone to implementing lexically-scoped functions, that is, `lambda` abstractions, which is the topic of Chapter 7.

6.1 The R_4 Language

The concrete and abstract syntax for function definitions and function application is shown in Figures 6.1 and 6.2, where we define the R_4 language. Programs in R_4 begin with zero or more function definitions. The function names from these definitions are in-scope for the entire program, including all other function definitions (so the ordering of function definitions does not matter). The concrete syntax for function application is (*exp exp* ...) where the first expression must evaluate to a function and the rest are the arguments. The abstract syntax for function application is `(Apply` *exp exp* ... `)`.

Functions are first-class in the sense that a function pointer is data and can be stored in memory or passed as a parameter to another function. Thus, we introduce a function type, written

`(`$type_1$ $\cdots$ $type_n$ `->` $type_r$`)`

for a function whose n parameters have the types $type_1$ through $type_n$ and whose return type is $type_r$. The main limitation of these functions (with respect to Racket functions) is that they are not lexically scoped. That is, the only external entities that can be referenced from inside a function body are other globally-defined functions. The syntax of R_4 prevents functions from being nested inside each other.

```
type  ::=  Integer | Boolean | (Vector type ...) | Void | (type ... -> type)
cmp   ::=  eq? | < | <= | > | >=
exp   ::=  int | (read) | (- exp) | (+ exp exp) | (- exp exp)
       |   var | (Let var exp exp)
       |   #t | #f | (and exp exp) | (or exp exp) | (not exp)
       |   (cmp exp exp) | (If exp exp exp)
       |   (vector exp ...) | (vector-ref exp int)
       |   (vector-set! exp int exp) | (void) | (has-type exp type)
       |   (exp exp ...)
def   ::=  (define (var [var:type] ...):type exp)
R4    ::=  def ... exp
```

Figure 6.1: The concrete syntax of R_4, extending R_3 (Figure 5.1).

```
exp  ::=  (Int int) | (Prim 'read '()) | (Prim '- (list exp ))
       |  (Prim '+ (list exp exp)) | (Prim '- (list exp exp))
       |  (Var var) | (Let var exp exp)
       |  (Bool bool) | (Prim 'and (list exp exp))
       |  (Prim 'or (list exp exp)) | (Prim 'not (list exp ))
       |  (Prim cmp (list exp exp)) | (If exp exp exp)
       |  (Prim 'vector (list exp*))
       |  (Prim 'vector-ref (list exp (Int int)))
       |  (Prim 'vector-set! (list exp (Int int) exp))
       |  (Void) | (HasType exp type) | (Apply exp exp ...)
def  ::=  (Def var ([var:type] ...) type '() exp)
R4   ::=  (ProgramDefsExp '() (def ...) exp)
```

Figure 6.2: The abstract syntax of R_4, extending R_3 (Figure 5.3).

```
(define (map-vec [f : (Integer -> Integer)]
                 [v : (Vector Integer Integer)])
       : (Vector Integer Integer)
  (vector (f (vector-ref v 0)) (f (vector-ref v 1))))

(define (add1 [x : Integer]) : Integer
  (+ x 1))

(vector-ref (map-vec add1 (vector 0 41)) 1)
```

Figure 6.3: Example of using functions in R_4.

The program in Figure 6.3 is a representative example of defining and using functions in R_4. We define a function `map-vec` that applies some other function `f` to both elements of a vector and returns a new vector containing the results. We also define a function `add1`. The program applies `map-vec` to `add1` and `(vector 0 41)`. The result is `(vector 1 42)`, from which we return the `42`.

The definitional interpreter for R_4 is in Figure 6.4. The case for the `ProgramDefsExp` form is responsible for setting up the mutual recursion between the top-level function definitions. We use the classic back-patching approach that uses mutable variables and makes two passes over the function definitions [Kelsey et al., 1998]. In the first pass we set up the top-level environment using a mutable cons cell for each function definition. Note that the `lambda` value for each function is incomplete; it does not yet include the environment. Once the top-level environment is constructed, we then iterate over it and update the `lambda` values to use the top-level environment.

The type checker for R_4 is is in Figure 6.5.

6.2 Functions in x86

The x86 architecture provides a few features to support the implementation of functions. We have already seen that x86 provides labels so that one can refer to the location of an instruction, as is needed for jump instructions. Labels can also be used to mark the beginning of the instructions for a function. Going further, we can obtain the address of a label by using the `leaq` instruction and PC-relative addressing. For example, the following puts the address of the `add1` label into the `rbx` register.

```
leaq add1(%rip), %rbx
```

```
(define (interp-exp env)
  (lambda (e)
    (define recur (interp-exp env))
    (match e
      ...
      [(Apply fun args)
       (define fun-val (recur fun))
       (define arg-vals (for/list ([e args]) (recur e)))
       (match fun-val
         [`(lambda (,xs ...) ,body ,fun-env)
          (define new-env (append (map cons xs arg-vals) fun-env))
          ((interp-exp new-env) body)])]
      ...
      )))

(define (interp-def d)
  (match d
    [(Def f (list `[,xs : ,ps] ...) rt _ body)
     (mcons f `(lambda ,xs ,body ()))]
    ))

(define (interp-R4 p)
  (match p
    [(ProgramDefsExp info ds body)
     (let ([top-level (for/list ([d ds]) (interp-def d))])
       (for/list ([b top-level])
         (set-mcdr! b (match (mcdr b)
                        [`(lambda ,xs ,body ())
                         `(lambda ,xs ,body ,top-level)])))
       ((interp-exp top-level) body))]
    ))
```

Figure 6.4: Interpreter for the R_4 language.

```
(define (fun-def-name d)
  (match d [(Def f (list `[,xs : ,ps] ...) rt info body) f]))

(define (fun-def-type d)
  (match d
    [(Def f (list `[,xs : ,ps] ...) rt info body) `(,@ps -> ,rt)]))

(define (type-check-exp env)
  (lambda (e)
    (match e
      ...
      [(Apply e es)
       (define-values (e^ ty) ((type-check-exp env) e))
       (define-values (e* ty*) (for/lists (e* ty*) ([e (in-list es)])
                                 ((type-check-exp env) e)))
       (match ty
         [`(,ty^* ... -> ,rt)
          (for ([arg-ty ty*] [prm-ty ty^*])
            (unless (equal? arg-ty prm-ty)
              (error "argument ~a not equal to parameter ~a" arg-ty prm-ty)))
          (values (HasType (Apply e^ e*) rt) rt)]
         [else (error "expected a function, not" ty)])])))

(define (type-check-def env)
  (lambda (e)
    (match e
      [(Def f (and p:t* (list `[,xs : ,ps] ...)) rt info body)
       (define new-env (append (map cons xs ps) env))
       (define-values (body^ ty^) ((type-check-exp new-env) body))
       (unless (equal? ty^ rt)
         (error "body type ~a not equal to return type ~a" ty^ rt))
       (Def f p:t* rt info body^)])))

(define (type-check env)
  (lambda (e)
    (match e
      [(ProgramDefsExp info ds body)
       (define new-env (for/list ([d ds])
                         (cons (fun-def-name d) (fun-def-type d))))
       (define ds^ (for/list ([d ds])
                     ((type-check-def new-env) d)))
       (define-values (body^ ty) ((type-check-exp new-env) body))
       (unless (equal? ty 'Integer)
         (error "result of the program must be an integer, not " ty))
       (ProgramDefsExp info ds^ body^)]
      [else (error 'type-check "R4/type-check unmatched ~a" e)])))
```

Figure 6.5: Type checker for the R_4 language.

The instruction pointer register `rip` (aka. the program counter) always points to the next instruction to be executed. When combined with an label, as in `add1(%rip)`, the linker computes the distance d between the address of `add1` and where the `rip` would be at that moment and then changes `add1(%rip)` to d`(%rip)`, which at runtime will compute the address of `add1`.

In Section 2.2 we used of the `callq` instruction to jump to a function whose location is given by a label. To support function calls in this chapter we instead will be jumping to a function whose location is given by an address in a register, that is, we need to make an *indirect function call*. The x86 syntax for this is a `callq` instruction but with an asterisk before the register name.

```
callq *%rbx
```

6.2.1 Calling Conventions

The `callq` instruction provides partial support for implementing functions: it pushes the return address on the stack and it jumps to the target. However, `callq` does not handle

1. parameter passing,
2. pushing frames on the procedure call stack and popping them off, or
3. determining how registers are shared by different functions.

These issues require coordination between the caller and the callee, which is often assembly code written by different programmers or generated by different compilers. As a result, people have developed *conventions* that govern how functions calls are performed. Here we use conventions that are compatible with those of the `gcc` compiler [Matz et al., 2013].

Regarding (1) parameter passing, the convention is to use the following six registers:

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

in that order, to pass arguments to a function. If there are more than six arguments, then the convention is to use space on the frame of the caller for the rest of the arguments. However, to ease the implementation of efficient tail calls (Section 6.2.2), we arrange to never need more than six arguments. The register `rax` is for the return value of the function.

Regarding (2) frames and the procedure call stack , recall from Section 2.2 that the stack grows down, with each function call using a chunk of space called a frame. The caller sets the stack pointer, register `rsp`, to the last data item in its frame. The callee must not change anything in the caller's frame, that is, anything that is at or above the stack pointer. The callee is free to use locations that are below the stack pointer.

Recall that we are storing variables of vector type on the root stack. So the prelude needs to move the root stack pointer `r15` up and the conclusion needs to move the root stack pointer back down. Also, the prelude must initialize to `0` this frame's slots in the root stack to signal to the garbage collector that those slots do not yet contain a pointer to a vector. Otherwise the garbage collector will interpret the garbage bits in those slots as memory addresses and try to traverse them, causing serious mayhem!

Regarding (3) the sharing of registers between different functions, recall from Section 3.1 that the registers are divided into two groups, the caller-saved registers and the callee-saved registers. The caller should assume that all the caller-saved registers get overwritten with arbitrary values by the callee. That is why we recommend in Section 3.1 that variables that are live during a function call should not be assigned to caller-saved registers.

On the flip side, if the callee wants to use a callee-saved register, the callee must save the contents of those registers on their stack frame and then put them back prior to returning to the caller. That is why we recommended in Section 3.1 that if the register allocator assigns a variable to a callee-saved register, then the prelude of the `main` function must save that register to the stack and the conclusion of `main` must restore it. This recommendation now generalizes to all functions.

Also recall that the base pointer, register `rbp`, is used as a point-of-reference within a frame, so that each local variable can be accessed at a fixed offset from the base pointer (Section 2.2). Figure 6.6 shows the general layout of the caller and callee frames.

6.2.2 Efficient Tail Calls

In general, the amount of stack space used by a program is determined by the longest chain of nested function calls. That is, if function f_1 calls f_2, f_2 calls f_3, ..., and f_{n-1} calls f_n, then the amount of stack space is bounded by $O(n)$. The depth n can grow quite large in the case of recursive or mutually recursive functions. However, in some cases we can arrange to use only constant space, i.e. $O(1)$, instead of $O(n)$.

If a function call is the last action in a function body, then that call is

Caller View	Callee View	Contents	Frame
8(%rbp)		return address	
0(%rbp)		old rbp	
-8(%rbp)		callee-saved 1	Caller
...		...	
$-8j$(%rbp)		callee-saved j	
$-8(j+1)$(%rbp)		local variable 1	
...		...	
$-8(j+k)$(%rbp)		local variable k	
	8(%rbp)	return address	
	0(%rbp)	old rbp	
	-8(%rbp)	callee-saved 1	Callee
	...	...	
	$-8n$(%rbp)	callee-saved n	
	$-8(n+1)$(%rbp)	local variable 1	
	...	...	
	$-8(n+m)$(%rsp)	local variable m	

Figure 6.6: Memory layout of caller and callee frames.

said to be a *tail call*. For example, in the following program, the recursive call to `tail-sum` is a tail call.

```
(define (tail-sum [n : Integer] [r : Integer]) : Integer
  (if (eq? n 0)
      r
      (tail-sum (- n 1) (+ n r))))

(+ (tail-sum 5 0) 27)
```

At a tail call, the frame of the caller is no longer needed, so we can pop the caller's frame before making the tail call. With this approach, a recursive function that only makes tail calls will only use $O(1)$ stack space. Functional languages like Racket typically rely heavily on recursive functions, so they typically guarantee that all tail calls will be optimized in this way.

However, some care is needed with regards to argument passing in tail calls. As mentioned above, for arguments beyond the sixth, the convention is to use space in the caller's frame for passing arguments. But for a tail call we pop the caller's frame and can no longer use it. Another alternative is to use space in the callee's frame for passing arguments. However, this option

is also problematic because the caller and callee's frame overlap in memory. As we begin to copy the arguments from their sources in the caller's frame, the target locations in the callee's frame might overlap with the sources for later arguments! We solve this problem by not using the stack for passing more than six arguments but instead using the heap, as we describe in the Section 6.5.

As mentioned above, for a tail call we pop the caller's frame prior to making the tail call. The instructions for popping a frame are the instructions that we usually place in the conclusion of a function. Thus, we also need to place such code immediately before each tail call. These instructions include restoring the callee-saved registers, so it is good that the argument passing registers are all caller-saved registers.

One last note regarding which instruction to use to make the tail call. When the callee is finished, it should not return to the current function, but it should return to the function that called the current one. Thus, the return address that is already on the stack is the right one, and we should not use `callq` to make the tail call, as that would unnecessarily overwrite the return address. Instead we can simply use the `jmp` instruction. Like the indirect function call, we write an *indirect jump* with a register prefixed with an asterisk. We recommend using `rax` to hold the jump target because the preceding conclusion overwrites just about everything else.

```
jmp *%rax
```

6.3 Shrink R_4

The `shrink` pass performs a minor modification to ease the later passes. This pass introduces an explicit `main` function and changes the top `ProgramDefsExp` form to `ProgramDefs` as follows.

`(ProgramDefsExp` *info* `(`*def* ...`)` *exp*`)`
⇒ `(ProgramDefs` *info* `(`*def* ... *mainDef*`))`

where *mainDef* is

`(Def 'main '() 'Integer '()` *exp*′`)`

6.4 Reveal Functions and the F_1 language

The syntax of R_4 is inconvenient for purposes of compilation in one respect: it conflates the use of function names and local variables. This is a problem

```
exp  ::= (Int int) | (Prim 'read '()) | (Prim '- (list exp ))
       | (Prim '+ (list exp exp)) | (Prim '- (list exp exp))
       | (Var var) | (Let var exp exp)
       | (Bool bool) | (Prim 'and (list exp exp))
       | (Prim 'or (list exp exp)) | (Prim 'not (list exp ))
       | (Prim cmp (list exp exp)) | (If exp exp exp)
       | (Prim 'vector (list exp*))
       | (Prim 'vector-ref (list exp (Int int)))
       | (Prim 'vector-set! (list exp (Int int) exp))
       | (Void) | (HasType exp type) | (Apply exp exp...)
       | (FunRef var)
def  ::= (Def var ([var:type]...) type '() exp)
F1   ::= (ProgramDefs '() (def...))
```

Figure 6.7: The abstract syntax F_1, an extension of R_4 (Figure 6.2).

because we need to compile the use of a function name differently than the use of a local variable; we need to use `leaq` to convert the function name (a label in x86) to an address in a register. Thus, it is a good idea to create a new pass that changes function references from just a symbol f to `(FunRef` f`)`. This pass is named `reveal-functions` and the output language, F_1, is defined in Figure 6.7.

Placing this pass after `uniquify` will make sure that there are no local variables and functions that share the same name. On the other hand, `reveal-functions` needs to come before the `explicate-control` pass because that pass helps us compile `FunRef` forms into assignment statements.

6.5 Limit Functions

Recall that we wish to limit the number of function parameters to six so that we do not need to use the stack for argument passing, which makes it easier to implement efficient tail calls. However, because the input language R_4 supports arbitrary numbers of function arguments, we have some work to do!

This pass transforms functions and function calls that involve more than six arguments to pass the first five arguments as usual, but it packs the rest of the arguments into a vector and passes it as the sixth argument.

Each function definition with too many parameters is transformed as

follows.

```
(Def f ([x1:T1] ... [xn:Tn]) Tr info body)
⇒
(Def f ([x1:T1] ... [x5:T5] [vec : (Vector T6 ... Tn)]) Tr info body′)
```

where the *body* is transformed into *body*′ by replacing the occurences of the later parameters with vector references.

```
(Var xi) ⇒ (Prim 'vector-ref (list vec (Int (i − 6))))
```

For function calls with too many arguments, the `limit-functions` pass transforms them in the following way.

```
(e0 e1 ... en)  ⇒  (e0 e1 ... e5 (vector e6 ... en))
```

6.6 Remove Complex Operators and Operands

The primary decisions to make for this pass is whether to classify `FunRef` and `Apply` as either simple or complex expressions. Recall that a simple expression will eventually end up as just an "immediate" argument of an x86 instruction. Function application will be translated to a sequence of instructions, so `Apply` must be classified as complex expression. Regarding `FunRef`, as discussed above, the function label needs to be converted to an address using the `leaq` instruction. Thus, even though `FunRef` seems rather simple, it needs to be classified as a complex expression so that we generate an assignment statement with a left-hand side that can serve as the target of the `leaq`.

6.7 Explicate Control and the C_3 language

Figures 6.8 and 6.9 define the concrete and abstract syntax for C_3, the output of `explicate-control`. The three mutually recursive functions for this pass, for assignment, tail, and predicate contexts, must all be updated with cases for `FunRef` and `Apply`. In assignment and predicate contexts, `Apply` becomes `Call` in C_3, whereas in tail position `Apply` becomes `TailCall` in C_3. We recommend defining a new function for processing function definitions. This code is similar to the case for `Program` in R_3. The top-level `explicate-control` function that handles the `ProgramDefs` form of R_4 can then apply this new function to all the function definitions.

```
atm    ::= int | var | #t | #f
cmp    ::= eq? | <
exp    ::= atm | (read) | (- atm) | (+ atm atm) | (not atm) | (cmp atm atm)
         | (allocate int type) | (vector-ref atm int)
         | (vector-set! atm int atm) | (global-value name) | (void)
         | (fun-ref label) | (call atm atm ...)
stmt   ::= (Assign var exp) | (Return exp) | (collect int)
tail   ::= (Return exp) | (seq stmt tail)
         | (goto label) | (If (cmp atm atm) (goto label) (goto label))
         | (tail-call atm atm ...)
def    ::= (define (label [var:type] ...):type ((label . tail) ...))
C3     ::= def ...
```

Figure 6.8: The C_3 language, extending C_2 (Figure 5.12) with functions.

```
atm    ::= (Int int) | (Var var) | (Bool bool)
cmp    ::= eq? | <
exp    ::= atm | (Prim 'read '())
         | (Prim '- (list atm )) | (Prim '+ (list atm atm))
         | (Prim not (list atm)) | (Prim cmp (list atm atm))
         | (Allocate int type)
         | (Prim 'vector-ref (list atm (Int int)))
         | (Prim 'vector-set! (list atm (Int int) atm))
         | (GlobalValue var) | (Void)
         | (FunRef label) | (Call atm atm ...)
stmt   ::= (Assign (Var var) exp) | (Collect int)
tail   ::= (Return exp) | (Seq stmt tail) | (Goto label)
         | (IfStmt (Prim cmp (list atm atm)) (Goto label) (Goto label))
         | (TailCall atm atm ...)
def    ::= (Def label ([var:type] ...) type ((label . tail) ...))
C3     ::= (ProgramDefs info (def ...))
```

Figure 6.9: The abstract syntax of C_3, extending C_2 (Figure 5.13).

arg	::=	(Int *int*) | (Reg *reg*) | (deref *reg int*)
	|	(byte-reg *reg*) | (global *name*)
	|	(fun-ref *label*)
cc	::=	e | l | le | g | ge
instr	::=	(addq *arg arg*) | (subq *arg arg*) | (negq *arg*) | (movq *arg arg*)
	|	(callq *label*) | (pushq *arg*) | (popq *arg*) | (retq)
	|	(xorq *arg arg*) | (cmpq *arg arg*) | (set*cc arg*)
	|	(movzbq *arg arg*) | (jmp *label*) | (j*cc label*) | (label *label*)
	|	(indirect-callq *arg*) | (tail-jmp *arg*)
	|	(leaq *arg arg*)
block	::=	(block *info instr* ...)
def	::=	(define (*label*) *info* ((*label* . *block*) ...))
$x86_3$	::=	(program *info def* ...)

Figure 6.10: The concrete syntax of $x86_3$ (extends $x86_2$ of Figure 5.16).

UNDER CONSTRUCTION

Figure 6.11: The abstract syntax of $x86_3$ (extends $x86_2$ of Figure 5.16).

6.8 Uncover Locals

The function for processing *tail* should be updated with a case for `TailCall`. We also recommend creating a new function for processing function definitions. Each function definition in C_3 has its own set of local variables, so the code for function definitions should be similar to the case for the `Program` form in C_2.

6.9 Select Instructions and the $x86_3$ Language

The output of select instructions is a program in the $x86_3$ language, whose syntax is defined in Figure 6.11.

An assignment of a function reference to a variable becomes a load-effective-address instruction as follows:

lhs `= (fun-ref` *f*`);` $\Rightarrow$ `leaq (fun-ref` *f*`),` *lhs*$'$

Regarding function definitions, we need to remove the parameters and instead perform parameter passing using the conventions discussed in Sec-

tion 6.2. That is, the arguments are passed in registers. We recommend turning the parameters into local variables and generating instructions at the beginning of the function to move from the argument passing registers to these local variables.

```
(Def f '([x1 : T1] [x2 : T2] ... ) Tr info G)
⇒
(Def f '() 'Integer info' G')
```

The G' control-flow graph is the same as G except that the `start` block is modified to add the instructions for moving from the argument registers to the parameter variables. So the `start` block of G shown on the left is changed to the code on the right.

```
start:                          start:
  instr1                          movq %rdi, x1
  ⋮                               movq %rsi, x2
  instrn                    ⇒     ⋮
                                  instr1
                                  ⋮
                                  instrn
```

By changing the parameters to local variables, we are giving the register allocator control over which registers or stack locations to use for them. If you implemented the move-biasing challenge (Section 3.6), the register allocator will try to assign the parameter variables to the corresponding argument register, in which case the `patch-instructions` pass will remove the `movq` instruction. Also, note that the register allocator will perform liveness analysis on this sequence of move instructions and build the interference graph. So, for example, x_1 will be marked as interfering with `rsi` and that will prevent the assignment of x_1 to `rsi`, which is good, because that would overwrite the argument that needs to move into x_2.

Next, consider the compilation of function calls. In the mirror image of handling the parameters of function definitions, the arguments need to be moved to the argument passing registers. The function call itself is performed with an indirect function call. The return value from the function is stored in `rax`, so it needs to be moved into the *lhs*.

```
lhs = (call fun arg1 arg2...));
⇒
movq arg1, %rdi
movq arg2, %rsi
```

```
⋮
callq *fun
movq %rax, lhs
```

Regarding tail calls, the parameter passing is the same as non-tail calls: generate instructions to move the arguments into to the argument passing registers. After that we need to pop the frame from the procedure call stack. However, we do not yet know how big the frame is; that gets determined during register allocation. So instead of generating those instructions here, we invent a new instruction that means "pop the frame and then do an indirect jump", which we name `TailJmp`.

Recall that in Section 2.6 we recommended using the label `start` for the initial block of a program, and in Section 2.7 we recommended labeling the conclusion of the program with `conclusion`, so that (`Return` *arg*) can be compiled to an assignment to `rax` followed by a jump to `conclusion`. With the addition of function definitions, we will have a starting block and conclusion for each function, but their labels need to be unique. We recommend prepending the function's name to `start` and `conclusion`, respectively, to obtain unique labels. (Alternatively, one could `gensym` labels for the start and conclusion and store them in the *info* field of the function definition.)

6.10 Uncover Live

The `IndirectCallq` instruction should be treated like `Callq` regarding its written locations W, in that they should include all the caller-saved registers. Recall that the reason for that is to force call-live variables to be assigned to callee-saved registers or to be spilled to the stack.

6.11 Build Interference Graph

With the addition of function definitions, we compute an interference graph for each function (not just one for the whole program).

Recall that in Section 5.9 we discussed the need to spill vector-typed variables that are live during a call to the `collect`. With the addition of functions to our language, we need to revisit this issue. Many functions will perform allocation and therefore have calls to the collector inside of them. Thus, we should not only spill a vector-typed variable when it is live during a call to `collect`, but we should spill the variable if it is live during any function call. Thus, in the `build-interference` pass, we recommend

adding interference edges between call-live vector-typed variables and the callee-saved registers (in addition to the usual addition of edges between call-live variables and the caller-saved registers).

6.12 Patch Instructions

In `patch-instructions`, you should deal with the x86 idiosyncrasy that the destination argument of `leaq` must be a register. Additionally, you should ensure that the argument of `TailJmp` is *rax*, our reserved register—this is to make code generation more convenient, because we will be trampling many registers before the tail call (as explained below).

6.13 Print x86

For the `print-x86` pass, we recommend the following translations:

`(FunRef` *label*`)` $\Rightarrow$ *label*`(%rip)`
`(IndirectCallq` *arg*`)` $\Rightarrow$ `callq *`*arg*

Handling `TailJmp` requires a bit more care. A straightforward translation of `TailJmp` would be `jmp *`*arg*, which is what we will want to do, but before the jump we need to pop the current frame. So we need to restore the state of the registers to the point they were at when the current function was called. This sequence of instructions is the same as the code for the conclusion of a function.

Note that your `print-x86` pass needs to add the code for saving and restoring callee-saved registers, if you have not already implemented that. This is necessary when generating code for function definitions.

6.14 An Example Translation

Figure 6.12 shows an example translation of a simple function in R_4 to x86. The figure also includes the results of the `explicate-control` and `select-instructions` passes. We have omitted the `HasType` AST nodes for readability.

Exercise 24. Expand your compiler to handle R_4 as outlined in this chapter. Create 5 new programs that use functions, including examples that pass functions and return functions from other functions and including recursive functions. Test your compiler on these new programs and all of your previously created test programs.

Figure 6.12: Example compilation of a simple function to x86.

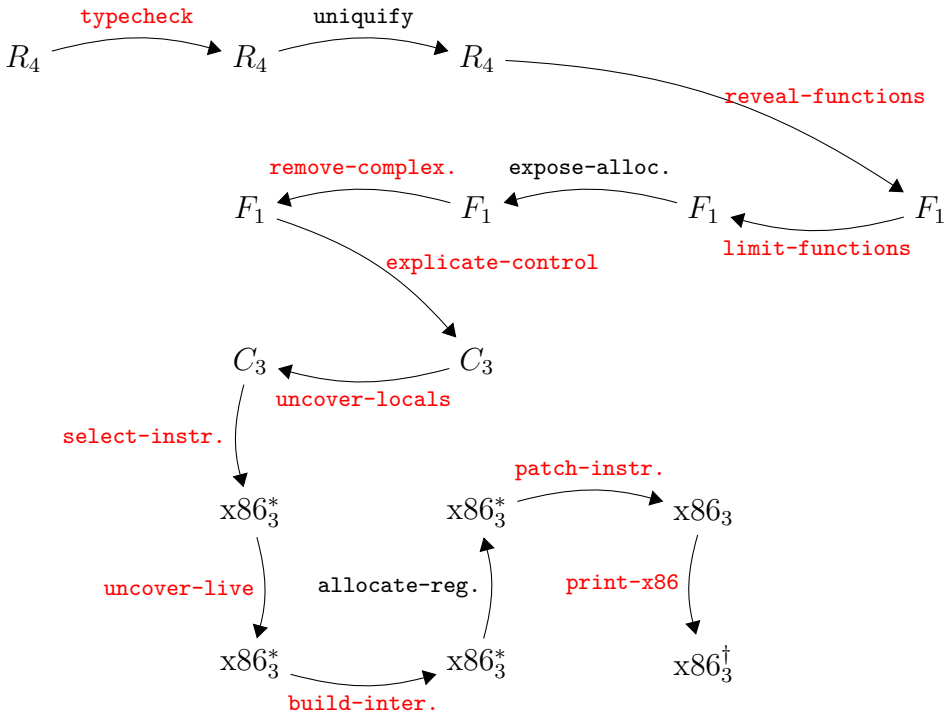


Figure 6.13: Diagram of the passes for R_4, a language with functions.

Figure 6.13 gives an overview of the passes needed for the compilation of R_4.

7

Lexically Scoped Functions

This chapter studies lexically scoped functions as they appear in functional languages such as Racket. By lexical scoping we mean that a function's body may refer to variables whose binding site is outside of the function, in an enclosing scope. Consider the example in Figure 7.1 written in R_5, which extends R_4 with anonymous functions using the `lambda` form. The body of the `lambda`, refers to three variables: `x`, `y`, and `z`. The binding sites for `x` and `y` are outside of the `lambda`. Variable `y` is bound by the enclosing `let` and `x` is a parameter of function `f`. The `lambda` is returned from the function `f`. The main expression of the program includes two calls to `f` with different arguments for `x`, first `5` then `3`. The functions returned from `f` are bound to variables `g` and `h`. Even though these two functions were created by the same `lambda`, they are really different functions because they use different values for `x`. Applying `g` to `11` produces `20` whereas applying `h` to `15` produces `22`. The result of this program is `42`.

The approach that we shall take for implementing lexically scoped func-

```
(define (f [x : Integer]) : (Integer -> Integer)
   (let ([y 4])
      (lambda: ([z : Integer]) : Integer
         (+ x (+ y z)))))

(let ([g (f 5)])
  (let ([h (f 3)])
    (+ (g 11) (h 15))))
```

Figure 7.1: Example of a lexically scoped function.

tions is to compile them into top-level function definitions, translating from R_5 into R_4. However, the compiler will need to provide special treatment for variable occurrences such as `x` and `y` in the body of the `lambda` of Figure 7.1. After all, an R_4 function may not refer to variables defined outside of it. To identify such variable occurrences, we review the standard notion of free variable.

Definition 25. *A variable is* free in expression *e if the variable occurs inside e but does not have an enclosing binding in e.*

For example, in the expression `(+ x (+ y z))` the variables `x`, `y`, and `z` are all free. On the other hand, only `x` and `y` are free in the following expression because `z` is bound by the `lambda`.

```
(lambda: ([z : Integer]) : Integer
  (+ x (+ y z)))
```

So the free variables of a `lambda` are the ones that will need special treatment. We need to arrange for some way to transport, at runtime, the values of those variables from the point where the `lambda` was created to the point where the `lambda` is applied. An efficient solution to the problem, due to Cardelli [1983], is to bundle into a vector the values of the free variables together with the function pointer for the lambda's code, an arrangement called a *flat closure* (which we shorten to just "closure"). Fortunately, we have all the ingredients to make closures, Chapter 5 gave us vectors and Chapter 6 gave us function pointers. The function pointer shall reside at index 0 and the values for the free variables will fill in the rest of the vector.

Let us revisit the example in Figure 7.1 to see how closures work. It's a three-step dance. The program first calls function `f`, which creates a closure for the `lambda`. The closure is a vector whose first element is a pointer to the top-level function that we will generate for the `lambda`, the second element is the value of `x`, which is `5`, and the third element is `4`, the value of `y`. The closure does not contain an element for `z` because `z` is not a free variable of the `lambda`. Creating the closure is step 1 of the dance. The closure is returned from `f` and bound to `g`, as shown in Figure 7.2. The second call to `f` creates another closure, this time with `3` in the second slot (for `x`). This closure is also returned from `f` but bound to `h`, which is also shown in Figure 7.2.

Continuing with the example, consider the application of `g` to `11` in Figure 7.1. To apply a closure, we obtain the function pointer in the first element of the closure and call it, passing in the closure itself and then the regular arguments, in this case `11`. This technique for applying a closure

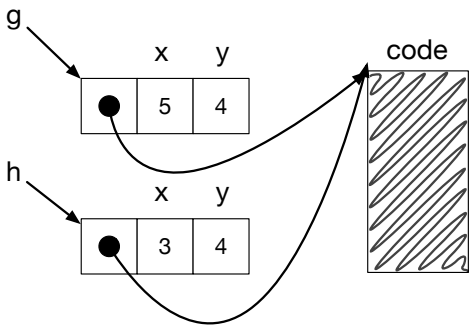


Figure 7.2: Example closure representation for the `lambda`'s in Figure 7.1.

is step 2 of the dance. But doesn't this `lambda` only take 1 argument, for parameter `z`? The third and final step of the dance is generating a top-level function for a `lambda`. We add an additional parameter for the closure and we insert a `let` at the beginning of the function for each free variable, to bind those variables to the appropriate elements from the closure parameter. This three-step dance is known as *closure conversion*. We discuss the details of closure conversion in Section 7.2 and the code generated from the example in Section 7.3. But first we define the syntax and semantics of R_5 in Section 7.1.

7.1 The R_5 Language

The concrete and abstract syntax for R_5, a language with anonymous functions and lexical scoping, is defined in Figures 7.3 and 7.4. It adds the `lambda` form to the grammar for R_4, which already has syntax for function application.

Figure 7.5 shows the definitional interpreter for R_5. The clause for `lambda` saves the current environment inside the returned `lambda`. Then the clause for `Apply` uses the environment from the `lambda`, the `lam-env`, when interpreting the body of the `lambda`. The `lam-env` environment is extended with the mapping of parameters to argument values.

Figure 7.6 shows how to type check the new `lambda` form. The body of the `lambda` is checked in an environment that includes the current environment (because it is lexically scoped) and also includes the `lambda`'s parameters. We require the body's type to match the declared return type.

```
type  ::=  Integer | Boolean | (Vector type ...) | Void | (type ... -> type)
exp   ::=  int | (read) | (- exp) | (+ exp exp) | (- exp exp)
       |   var | (Let var exp exp)
       |   #t | #f | (and exp exp) | (or exp exp) | (not exp)
       |   (eq? exp exp) | (If exp exp exp)
       |   (vector exp ...) | (vector-ref exp int)
       |   (vector-set! exp int exp) | (void) | (exp exp ...)
       |   (lambda: ([var : type] ...) : type exp)
def   ::=  (define (var [var : type] ...) : type exp)
R5    ::=  (program def ... exp)
```

Figure 7.3: Concrete syntax of R_5, extending R_4 (Figure 6.2) with `lambda`.

```
exp   ::=  (Int int) | (Prim 'read '()) | (Prim '- (list exp ))
       |   (Prim '+ (list exp exp)) | (Prim '- (list exp exp))
       |   (Var var) | (Let var exp exp)
       |   (Bool bool) | (Prim 'and (list exp exp))
       |   (Prim 'or (list exp exp)) | (Prim 'not (list exp ))
       |   (Prim cmp (list exp exp)) | (If exp exp exp)
       |   (Prim 'vector (list exp*))
       |   (Prim 'vector-ref (list exp (Int int)))
       |   (Prim 'vector-set! (list exp (Int int) exp))
       |   (Void) | (HasType exp type) | (Apply exp exp ...)
       |   (Lambda [var : type] ... type exp)
def   ::=  (Def var ([var : type] ...) type '() exp)
R5    ::=  (ProgramDefsExp '() (def ...) exp)
```

Figure 7.4: The abstract syntax of R_5, extending R_4 (Figure 6.2).

```
(define (interp-exp env)
  (lambda (e)
    (define recur (interp-exp env))
    (match e
      ...
      [(Lambda (list `[,xs : ,Ts] ...) rT body)
       `(lambda ,xs ,body ,env)]
      [(Apply fun args)
       (define fun-val ((interp-exp env) fun))
       (define arg-vals (map (interp-exp env) args))
       (match fun-val
         [`(lambda ,xs ,body ,lam-env)
          (define new-env (append (map cons xs arg-vals) lam-env))
          ((interp-exp new-env) body)]
         [else (error "interp-exp, expected function, not" fun-val)])]
      [else (error 'interp-exp "unrecognized expression")]
      )))
```

Figure 7.5: Interpreter for R_5.

```
(define (typecheck-R5 env)
  (lambda (e)
    (match e
      [(Lambda (and bnd `([,xs : ,Ts] ...)) rT body)
       (define-values (new-body bodyT)
          ((type-check-exp (append (map cons xs Ts) env)) body))
       (define ty `(,@Ts -> ,rT))
       (cond
         [(equal? rT bodyT)
          (values (HasType (Lambda bnd rT new-body) ty) ty)]
         [else
          (error "mismatch in return type" bodyT rT)])]
      ...
      )))
```

Figure 7.6: Type checking the `lambda`'s in R_5.

7.2 Closure Conversion

The compiling of lexically-scoped functions into top-level function definitions is accomplished in the pass `convert-to-closures` that comes after `reveal-functions` and before `limit-functions`.

As usual, we shall implement the pass as a recursive function over the AST. All of the action is in the clauses for `lambda` and `Apply`. We transform a `lambda` expression into an expression that creates a closure, that is, creates a vector whose first element is a function pointer and the rest of the elements are the free variables of the `lambda`. The *name* is a unique symbol generated to identify the function.

```
(lambda: (ps ...) : rt body)  ⇒  (vector name fvs ...)
```

In addition to transforming each `lambda` into a `vector`, we must create a top-level function definition for each `lambda`, as shown below.

```
(define (name [clos : (Vector _ fvts ...)] ps ...)
  (let ([fvs1 (vector-ref clos 1)])
    ...
    (let ([fvsn (vector-ref clos n)])
      body′)...))
```

The `clos` parameter refers to the closure. The *ps* parameters are the normal parameters of the `lambda`. The types *fvts* are the types of the free variables in the lambda and the underscore is a dummy type because it is rather difficult to give a type to the function in the closure's type, and it does not matter. The sequence of `let` forms bind the free variables to their values obtained from the closure.

We transform function application into code that retrieves the function pointer from the closure and then calls the function, passing in the closure as the first argument. We bind e' to a temporary variable to avoid code duplication.

```
(app e es ...)  ⇒  (let ([tmp e′])
                     (app (vector-ref tmp 0) tmp es′))
```

There is also the question of what to do with top-level function definitions. To maintain a uniform translation of function application, we turn function references into closures.

```
(fun-ref f)  ⇒  (vector (fun-ref f))
```

The top-level function definitions need to be updated as well to take an extra closure parameter.

7.3 An Example Translation

Figure 7.7 shows the result of closure conversion for the example program demonstrating lexical scoping that we discussed at the beginning of this chapter.

```
(define (f [x : Integer]) : (Integer -> Integer)
   (let ([y 4])
      (lambda: ([z : Integer]) : Integer
         (+ x (+ y z)))))
(let ([g (f 5)])
  (let ([h (f 3)])
    (+ (g 11) (h 15)))))
```

⇓

```
(define (f (x : Integer)) : (Integer -> Integer)
  (let ((y 4))
     (lambda: ((z : Integer)) : Integer
       (+ x (+ y z)))))
 (let ((g (app (fun-ref f) 5)))
    (let ((h (app (fun-ref f) 3)))
      (+ (app g 11) (app h 15)))))
```

⇓

```
(define (f (clos.1 : _) (x : Integer)) : (Integer -> Integer)
   (let ((y 4))
      (vector (fun-ref lam.1) x y)))
(define (lam.1 (clos.2 : _) (z : Integer)) : Integer
   (let ((x (vector-ref clos.2 1)))
      (let ((y (vector-ref clos.2 2)))
         (+ x (+ y z)))))
 (let ((g (let ((t.1 (vector (fun-ref f))))
            (app (vector-ref t.1 0) t.1 5))))
    (let ((h (let ((t.2 (vector (fun-ref f))))
               (app (vector-ref t.2 0) t.2 3))))
       (+ (let ((t.3 g)) (app (vector-ref t.3 0) t.3 11))
          (let ((t.4 h)) (app (vector-ref t.4 0) t.4 15))))))
```

Figure 7.7: Example of closure conversion.

Figure 7.8 provides an overview of all the passes needed for the compilation of R_5.

Exercise 26. Expand your compiler to handle R_5 as outlined in this chap-

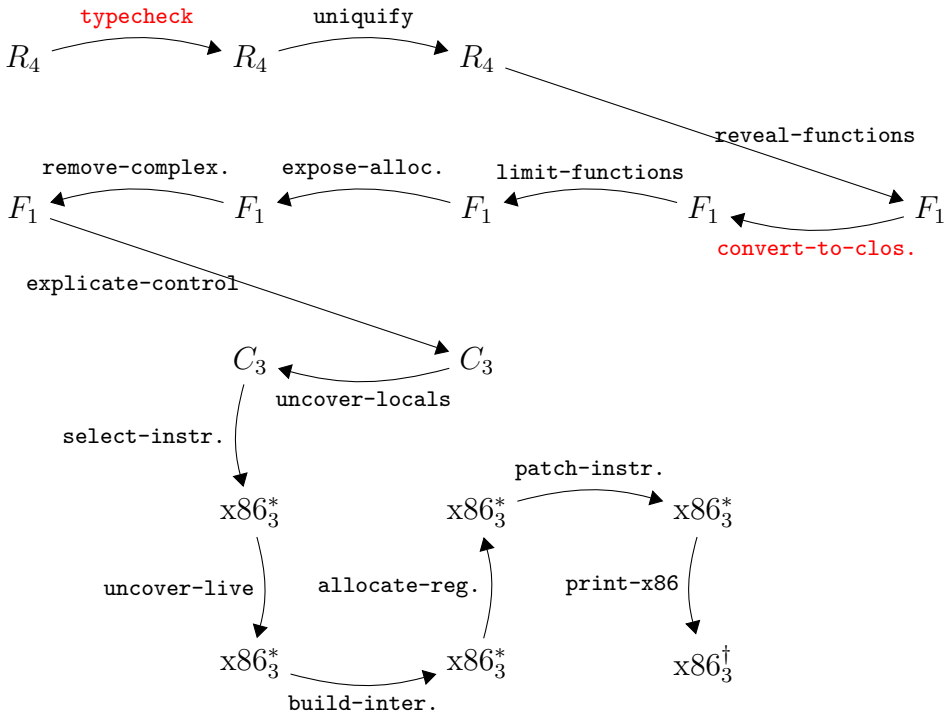


Figure 7.8: Diagram of the passes for R_5, a language with lexically-scoped functions.

ter. Create 5 new programs that use `lambda` functions and make use of lexical scoping. Test your compiler on these new programs and all of your previously created test programs.

8

Dynamic Typing

In this chapter we discuss the compilation of a dynamically typed language, named R_7, that is a subset of the Racket language. (Recall that in the previous chapters we have studied subsets of the *Typed* Racket language.) In dynamically typed languages, an expression may produce values of differing type. Consider the following example with a conditional expression that may return a Boolean or an integer depending on the input to the program.

```
(not (if (eq? (read) 1) #f 0))
```

Languages that allow expressions to produce different kinds of values are called *polymorphic*. There are many kinds of polymorphism, such as subtype polymorphism and parametric polymorphism [Cardelli and Wegner, 1985]. The kind of polymorphism we are talking about here does not have a special name, but it is the usual kind that arises in dynamically typed languages.

Another characteristic of dynamically typed languages is that primitive operations, such as `not`, are often defined to operate on many different types of values. In fact, in Racket, the `not` operator produces a result for any kind of value: given `#f` it returns `#t` and given anything else it returns `#f`. Furthermore, even when primitive operations restrict their inputs to values of a certain type, this restriction is enforced at runtime instead of during compilation. For example, the following vector reference results in a run-time contract violation.

```
(vector-ref (vector 42) #t)
```

The syntax of R_7, our subset of Racket, is defined in Figure 8.1. The definitional interpreter for R_7 is given in Figure 8.2.

Let us consider how we might compile R_7 to x86, thinking about the first example above. Our bit-level representation of the Boolean `#f` is zero and

$$
\begin{array}{rcl}
\mathit{cmp} & ::= & \texttt{eq?} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \\
\mathit{exp} & ::= & \mathit{int} \mid (\texttt{read}) \mid (\texttt{-}\ \mathit{exp}) \mid (\texttt{+}\ \mathit{exp}\ \mathit{exp}) \mid (\texttt{-}\ \mathit{exp}\ \mathit{exp}) \\
& \mid & \mathit{var} \mid (\texttt{Let}\ \mathit{var}\ \mathit{exp}\ \mathit{exp}) \\
& \mid & \texttt{\#t} \mid \texttt{\#f} \mid (\texttt{and}\ \mathit{exp}\ \mathit{exp}) \mid (\texttt{or}\ \mathit{exp}\ \mathit{exp}) \mid (\texttt{not}\ \mathit{exp}) \\
& \mid & (\mathit{cmp}\ \mathit{exp}\ \mathit{exp}) \mid (\texttt{If}\ \mathit{exp}\ \mathit{exp}\ \mathit{exp}) \\
& \mid & (\texttt{vector}\ \mathit{exp}\ldots) \mid (\texttt{vector-ref}\ \mathit{exp}\ \mathit{exp}) \\
& \mid & (\texttt{vector-set!}\ \mathit{exp}\ \mathit{exp}\ \mathit{exp}) \mid (\texttt{void}) \\
& \mid & (\mathit{exp}\ \mathit{exp}\ldots) \mid (\texttt{lambda}\ (\mathit{var}\ldots)\ \mathit{exp}) \\
& \mid & (\texttt{boolean?}\ \mathit{exp}) \mid (\texttt{integer?}\ \mathit{exp}) \\
& \mid & (\texttt{vector?}\ \mathit{exp}) \mid (\texttt{procedure?}\ \mathit{exp}) \mid (\texttt{void?}\ \mathit{exp}) \\
\mathit{def} & ::= & (\texttt{define}\ (\mathit{var}\ \mathit{var}\ldots)\ \mathit{exp}) \\
R_7 & ::= & (\texttt{program}\ \mathit{def}\ldots\ \mathit{exp})
\end{array}
$$

Figure 8.1: Syntax of R_7, an untyped language (a subset of Racket).

similarly for the integer `0`. However, `(not #f)` should produce `#t` whereas `(not 0)` should produce `#f`. Furthermore, the behavior of `not`, in general, cannot be determined at compile time, but depends on the runtime type of its input, as in the example above that depends on the result of `(read)`.

The way around this problem is to include information about a value's runtime type in the value itself, so that this information can be inspected by operators such as `not`. In particular, we shall steal the 3 right-most bits from our 64-bit values to encode the runtime type. We shall use 001 to identify integers, 100 for Booleans, 010 for vectors, 011 for procedures, and 101 for the void value. We shall refer to these 3 bits as the *tag* and we define the following auxiliary function.

$$
\begin{aligned}
\mathit{tagof}(\texttt{Integer}) &= 001 \\
\mathit{tagof}(\texttt{Boolean}) &= 100 \\
\mathit{tagof}((\texttt{Vector}\ldots)) &= 010 \\
\mathit{tagof}((\texttt{Vectorof}\ldots)) &= 010 \\
\mathit{tagof}((\ldots\texttt{->}\ldots)) &= 011 \\
\mathit{tagof}(\texttt{Void}) &= 101
\end{aligned}
$$

(We shall say more about the new `Vectorof` type shortly.) This stealing of 3 bits comes at some price: our integers are reduced to ranging from -2^{60} to 2^{60}. The stealing does not adversely affect vectors and procedures because those values are addresses, and our addresses are 8-byte aligned so the rightmost 3 bits are unused, they are always 000. Thus, we do not lose

```
(define (get-tagged-type v) (match v [`(tagged ,v1 ,ty) ty]))

(define (valid-op? op) (member op '(+ - and or not)))

(define (interp-r7 env)
  (lambda (ast)
    (define recur (interp-r7 env))
    (match ast
      [(? symbol?) (lookup ast env)]
      [(? integer?) `(inject ,ast Integer)]
      [#t `(inject #t Boolean)]
      [#f `(inject #f Boolean)]
      [`(read) `(inject ,(read-fixnum) Integer)]
      [`(lambda (,xs ...) ,body)
       `(inject (lambda ,xs ,body ,env) (,@(map (lambda (x) 'Any) xs) -> Any))]
      [`(define (,f ,xs ...) ,body)
       (mcons f `(lambda ,xs ,body))]
      [`(program ,ds ... ,body)
       (let ([top-level (for/list ([d ds]) ((interp-r7 '()) d))])
         (for/list ([b top-level])
           (set-mcdr! b (match (mcdr b)
                          [`(lambda ,xs ,body)
                           `(inject (lambda ,xs ,body ,top-level)
                                    (,@(map (lambda (x) 'Any) xs) -> Any))])))
         ((interp-r7 top-level) body))]
      [`(vector ,(app recur elts) ...)
       (define tys (map get-tagged-type elts))
       `(inject ,(apply vector elts) (Vector ,@tys))]
      [`(vector-set! ,(app recur v1) ,n ,(app recur v2))
         (match v1
           [`(inject ,vec ,ty)
            (vector-set! vec n v2)
           `(inject (void) Void)])]
      [`(vector-ref ,(app recur v) ,n)
       (match v [`(inject ,vec ,ty) (vector-ref vec n)])]
      [`(let ([,x ,(app recur v)]) ,body)
       ((interp-r7 (cons (cons x v) env)) body)]
      [`(,op ,es ...) #:when (valid-op? op)
       (interp-r7-op op (for/list ([e es]) (recur e)))]
      [`(eq? ,(app recur l) ,(app recur r))
       `(inject ,(equal? l r) Boolean)]
      [`(if ,(app recur q) ,t ,f)
       (match q
         [`(inject #f Boolean) (recur f)]
         [else (recur t)])]
      [`(,(app recur f-val) ,(app recur vs) ...)
       (match f-val
         [`(inject (lambda (,xs ...) ,body ,lam-env) ,ty)
          (define new-env (append (map cons xs vs) lam-env))
          ((interp-r7 new-env) body)]
         [else (error "interp-r7, expected function, not" f-val)])])))
```

Figure 8.2: Interpreter for the R_7 language. UPDATE ME -Jeremy

information by overwriting the rightmost 3 bits with the tag and we can simply zero-out the tag to recover the original address.

In some sense, these tagged values are a new kind of value. Indeed, we can extend our *typed* language with tagged values by adding a new type to classify them, called `Any`, and with operations for creating and using tagged values, yielding the R_6 language that we define in Section 8.1. The R_6 language provides the fundamental support for polymorphism and runtime types that we need to support dynamic typing.

There is an interesting interaction between tagged values and garbage collection. A variable of type `Any` might refer to a vector and therefore it might be a root that needs to be inspected and copied during garbage collection. Thus, we need to treat variables of type `Any` in a similar way to variables of type `Vector` for purposes of register allocation, which we discuss in Section 8.4. One concern is that, if a variable of type `Any` is spilled, it must be spilled to the root stack. But this means that the garbage collector needs to be able to differentiate between (1) plain old pointers to tuples, (2) a tagged value that points to a tuple, and (3) a tagged value that is not a tuple. We enable this differentiation by choosing not to use the tag 000. Instead, that bit pattern is reserved for identifying plain old pointers to tuples. On the other hand, if one of the first three bits is set, then we have a tagged value, and inspecting the tag can differentiation between vectors (010) and the other kinds of values.

We shall implement our untyped language R_7 by compiling it to R_6 (Section 8.5), but first we describe the how to extend our compiler to handle the new features of R_6 (Sections 8.2, 8.3, and 8.4).

8.1 The R_6 Language: Typed Racket + `Any`

The syntax of R_6 is defined in Figure 8.3. The (`inject` e T) form converts the value produced by expression e of type T into a tagged value. The (`project` e T) form converts the tagged value produced by expression e into a value of type T or else halts the program if the type tag is equivalent to T. We treat (`Vectorof Any`) as equivalent to (`Vector Any` ...).

Note that in both `inject` and `project`, the type T is restricted to the flat types *ftype*, which simplifies the implementation and corresponds with what is needed for compiling untyped Racket. The type predicates, (`boolean?` e) etc., expect a tagged value and return `#t` if the tag corresponds to the predicate, and return `#t` otherwise. Selections from the type checker for R_6 are shown in Figure 8.4 and the interpreter for R_6 is in Figure 8.5.

type	::=	`Integer` | `Boolean` | (`Vector` *type* ...) | (`Vectorof` *type*) | `Void`
	|	(*type* ... `->` *type*) | `Any`
ftype	::=	`Integer` | `Boolean` | `Void` | (`Vectorof Any`) | (`Vector Any` ...)
	|	(`Any` ... `->` `Any`)
cmp	::=	`eq?` | `<` | `<=` | `>` | `>=`
exp	::=	*int* | (`read`) | (`-` *exp*) | (`+` *exp exp*) | (`-` *exp exp*)
	|	*var* | (`Let` *var exp exp*)
	|	`#t` | `#f` | (`and` *exp exp*) | (`or` *exp exp*) | (`not` *exp*)
	|	(*cmp exp exp*) | (`If` *exp exp exp*)
	|	(`vector` *exp* ...) | (`vector-ref` *exp int*)
	|	(`vector-set!` *exp int exp*) | (`void`)
	|	(*exp exp* ...) | (`lambda:` ([*var* : *type*] ...) : *type exp*)
	|	(`inject` *exp ftype*) | (`project` *exp ftype*)
	|	(`boolean?` *exp*) | (`integer?` *exp*)
	|	(`vector?` *exp*) | (`procedure?` *exp*) | (`void?` *exp*)
def	::=	(`define` (*var* [*var* : *type*] ...) : *type exp*)
R_6	::=	(`program` *def* ... *exp*)

Figure 8.3: Syntax of R_6, extending R_5 (Figure 7.4) with `Any`.

```
(define (flat-ty? ty) ...)

(define (typecheck-R6 env)
  (lambda (e)
    (define recur (typecheck-R6 env))
    (match e
       [`(inject ,e ,ty)
        (unless (flat-ty? ty)
          (error "may only inject a value of flat type, not ~a" ty))
        (define-values (new-e e-ty) (recur e))
        (cond
         [(equal? e-ty ty)
          (values `(inject ,new-e ,ty) 'Any)]
         [else
          (error "inject expected ~a to have type ~a" e ty)])]
      [`(project ,e ,ty)
       (unless (flat-ty? ty)
         (error "may only project to a flat type, not ~a" ty))
       (define-values (new-e e-ty) (recur e))
       (cond
        [(equal? e-ty 'Any)
         (values `(project ,new-e ,ty) ty)]
        [else
         (error "project expected ~a to have type Any" e)])]
      [`(vector-ref ,e ,i)
       (define-values (new-e e-ty) (recur e))
       (match e-ty
         [`(Vector ,ts ...) ...]
         [`(Vectorof ,ty)
          (unless (exact-nonnegative-integer? i)
            (error 'type-check "invalid index ~a" i))
          (values `(vector-ref ,new-e ,i) ty)]
         [else (error "expected a vector in vector-ref, not" e-ty)])]
       ...
     )))
```

Figure 8.4: Type checker for parts of the R_6 language.

```
(define primitives (set 'boolean? ...))

(define (interp-op op)
  (match op
     ['boolean? (lambda (v)
                  (match v
                     [`(tagged ,v1 Boolean) #t]
                     [else #f]))]
     ...))

;; Equivalence of flat types
(define (tyeq? t1 t2)
  (match `(,t1 ,t2)
    [`((Vectorof Any) (Vector ,t2s ...))
     (for/and ([t2 t2s]) (eq? t2 'Any))]
    [`((Vector ,t1s ...) (Vectorof Any))
     (for/and ([t1 t1s]) (eq? t1 'Any))]
    [else (equal? t1 t2)]))

(define (interp-R6 env)
  (lambda (ast)
    (match ast
       [`(inject ,e ,t)
        `(tagged ,((interp-R6 env) e) ,t)]
       [`(project ,e ,t2)
        (define v ((interp-R6 env) e))
        (match v
           [`(tagged ,v1 ,t1)
            (cond [(tyeq? t1 t2)
                   v1]
                  [else
                   (error "in project, type mismatch" t1 t2)])]
           [else
            (error "in project, expected tagged value" v)])]
       ...)))
```

Figure 8.5: Interpreter for R_6.

8.2 Shrinking R_6

In the `shrink` pass we recommend compiling `project` into an explicit `if` expression that uses three new operations: `tag-of-any`, `value-of-any`, and `exit`. The `tag-of-any` operation retrieves the type tag from a tagged value of type `Any`. The `value-of-any` retrieves the underlying value from a tagged value. Finally, the `exit` operation ends the execution of the program by invoking the operating system's `exit` function. So the translation for `project` is as follows. (We have omitted the `has-type` AST nodes to make this output more readable.)

```
(project e type)  ⇒  (let ([tmp e'])
                       (if (eq? (tag-of-any tmp) tag)
                           (value-of-any tmp)
                           (exit)))
```

Similarly, we recommend translating the type predicates (`boolean?`, etc.) into uses of `tag-of-any` and `eq?`.

8.3 Instruction Selection for R_6

Inject We recommend compiling an `inject` as follows if the type is `Integer` or `Boolean`. The `salq` instruction shifts the destination to the left by the number of bits specified its source argument (in this case 3, the length of the tag) and it preserves the sign of the integer. We use the `orq` instruction to combine the tag and the value to form the tagged value.

```
(assign lhs (inject e T))  ⇒  (movq e' lhs')
                              (salq (int 3) lhs')
                              (orq (int tagof(T)) lhs')
```

The instruction selection for vectors and procedures is different because their is no need to shift them to the left. The rightmost 3 bits are already zeros as described above. So we just combine the value and the tag using `orq`.

```
(assign lhs (inject e T))  ⇒  (movq e' lhs')
                              (orq (int tagof(T)) lhs')
```

Tag of Any Recall that the `tag-of-any` operation extracts the type tag from a value of type `Any`. The type tag is the bottom three bits, so we obtain the tag by taking the bitwise-and of the value with 111 (7 in decimal).

```
(assign lhs (tag-of-any e))    ⇒  (movq e' lhs')
                                  (andq (int 7) lhs')
```

Value of Any Like `inject`, the instructions for `value-of-any` are different depending on whether the type T is a pointer (vector or procedure) or not (Integer or Boolean). The following shows the instruction selection for Integer and Boolean. We produce an untagged value by shifting it to the right by 3 bits.

```
(assign lhs (project e T))    ⇒  (movq e' lhs')
                                 (sarq (int 3) lhs')
```

In the case for vectors and procedures, there is no need to shift. Instead we just need to zero-out the rightmost 3 bits. We accomplish this by creating the bit pattern ...0111 (7 in decimal) and apply `bitwise-not` to obtain ...1000 which we `movq` into the destination *lhs*. We then generate `andq` with the tagged value to get the desired result.

```
(assign lhs (project e T))    ⇒  (movq (int ...1000) lhs')
                                 (andq e' lhs')
```

8.4 Register Allocation for R_6

As mentioned above, a variable of type `Any` might refer to a vector. Thus, the register allocator for R_6 needs to treat variable of type `Any` in the same way that it treats variables of type `Vector` for purposes of garbage collection. In particular,

- If a variable of type `Any` is live during a function call, then it must be spilled. One way to accomplish this is to augment the pass `build-interference` to mark all variables that are live after a `callq` as interfering with all the registers.

- If a variable of type `Any` is spilled, it must be spilled to the root stack instead of the normal procedure call stack.

Exercise 27. Expand your compiler to handle R_6 as discussed in the last few sections. Create 5 new programs that use the `Any` type and the new operations (`inject`, `project`, `boolean?`, etc.). Test your compiler on these new programs and all of your previously created test programs.

8.5 Compiling R_7 to R_6

Figure 8.6 shows the compilation of many of the R_7 forms into R_6. An important invariant of this pass is that given a subexpression e of R_7, the pass will produce an expression e' of R_6 that has type `Any`. For example, the first row in Figure 8.6 shows the compilation of the Boolean `#t`, which must be injected to produce an expression of type `Any`. The second row of Figure 8.6, the compilation of addition, is representative of compilation for many operations: the arguments have type `Any` and must be projected to `Integer` before the addition can be performed.

The compilation of `lambda` (third row of Figure 8.6) shows what happens when we need to produce type annotations: we simply use `Any`. The compilation of `if` and `eq?` demonstrate how this pass has to account for some differences in behavior between R_7 and R_6. The R_7 language is more permissive than R_6 regarding what kind of values can be used in various places. For example, the condition of an `if` does not have to be a Boolean. For `eq?`, the arguments need not be of the same type (but in that case, the result will be `#f`).

Exercise 28. Expand your compiler to handle R_7 as outlined in this chapter. Create tests for R_7 by adapting all of your previous test programs by removing type annotations. Add 5 more tests programs that specifically rely on the language being dynamically typed. That is, they should not be legal programs in a statically typed language, but nevertheless, they should be valid R_7 programs that run to completion without error.

`#t`	⇒	`(inject #t Boolean)`

`(+` e_1 e_2`)`	⇒	`(inject`
		`  (+ (project` e'_1 `Integer)`
		`  (project` e'_2 `Integer))`
		`  Integer)`

`(lambda (`$x_1 \ldots$`)` e`)`	⇒	`(inject (lambda: ([`x_1`:Any]...):Any` e'`)` `  (Any...Any -> Any))`
`(app` e_0 $e_1 \ldots e_n$`)`	⇒	`(app (project` e'_0 `(Any...Any -> Any))` $e'_1 \ldots e'_n$`)`

`(vector-ref` e_1 e_2`)`	⇒	`(let ([tmp1 (project` e'_1 `(Vectorof Any))])`
		`  (let ([tmp2 (project` e'_2 `Integer)])`
		`  (vector-ref tmp1 tmp2)))`

`(if` e_1 e_2 e_3`)`	⇒	`(if (eq?` e'_1 `(inject #f Boolean))` e'_3 e'_2`)`
`(eq?` e_1 e_2`)`	⇒	`(inject (eq?` e'_1 e'_2`) Boolean)`

Figure 8.6: Compiling R_7 to R_6.

9

Gradual Typing

This chapter will be based on the ideas of Siek and Taha [2006].

10

Parametric Polymorphism

This chapter may be based on ideas from Cardelli [1984], Leroy [1992], Shao [1997], or Harper and Morrisett [1995].

11

High-level Optimization

This chapter will present a procedure inlining pass based on the algorithm of Waddell and Dybvig [1997].

12

Appendix

12.1 Interpreters

We provide interpreters for each of the source languages R_0, R_1, ... in the files `interp-R1.rkt`, `interp-R2.rkt`, etc. The interpreters for the intermediate languages C_0 and C_1 are in `interp-C0.rkt` and `interp-C1.rkt`. The interpreters for the rest of the intermediate languages, including pseudo-x86 and x86 are in the `interp.rkt` file.

12.2 Utility Functions

The utility functions described here are in the `utilities.rkt` file.

`interp-tests` The `interp-tests` function runs the compiler passes and the interpreters on each of the specified tests to check whether each pass is correct. The `interp-tests` function has the following parameters:

name (a string) a name to identify the compiler,

typechecker a function of exactly one argument that either raises an error using the `error` function when it encounters a type error, or returns `#f` when it encounters a type error. If there is no type error, the type checker returns the program.

passes a list with one entry per pass. An entry is a list with three things: a string giving the name of the pass, the function that implements the pass (a translator from AST to AST), and a function that implements the interpreter (a function from AST to result value) for the language of the output of the pass.

source-interp an interpreter for the source language. The interpreters from Appendix 12.1 make a good choice.

test-family (a string) for example, `"r1"`, `"r2"`, etc.

tests a list of test numbers that specifies which tests to run. (see below)

The `interp-tests` function assumes that the subdirectory `tests` has a collection of Racket programs whose names all start with the family name, followed by an underscore and then the test number, ending with the file extension `.rkt`. Also, for each test program that calls `read` one or more times, there is a file with the same name except that the file extension is `.in` that provides the input for the Racket program. If the test program is expected to fail type checking, then there should be an empty file of the same name but with extension `.tyerr`.

`compiler-tests` runs the compiler passes to generate x86 (a `.s` file) and then runs the GNU C compiler (gcc) to generate machine code. It runs the machine code and checks that the output is 42. The parameters to the `compiler-tests` function are similar to those of the `interp-tests` function, and consist of

- a compiler name (a string),
- a type checker,
- description of the passes,
- name of a test-family, and
- a list of test numbers.

`compile-file` takes a description of the compiler passes (see the comment for `interp-tests`) and returns a function that, given a program file name (a string ending in `.rkt`), applies all of the passes and writes the output to a file whose name is the same as the program file name but with `.rkt` replaced with `.s`.

`read-program` takes a file path and parses that file (it must be a Racket program) into an abstract syntax tree.

`parse-program` takes an S-expression representation of an abstract syntax tree and converts it into the struct-based representation.

`assert` takes two parameters, a string (`msg`) and Boolean (`bool`), and displays the message `msg` if the Boolean `bool` is false.

`lookup` takes a key and an alist, and returns the first value that is associated with the given key, if there is one. If not, an error is triggered. The alist may contain both immutable pairs (built with `cons`) and mutable pairs (built with `mcons`).

12.3 x86 Instruction Set Quick-Reference

Table 12.1 lists some x86 instructions and what they do. We write $A \to B$ to mean that the value of A is written into location B. Address offsets are given in bytes. The instruction arguments A, B, C can be immediate constants (such as `$4`), registers (such as `%rax`), or memory references (such as `-4(%ebp)`). Most x86 instructions only allow at most one memory reference per instruction. Other operands must be immediates or registers.

Instruction	**Operation**
`addq` A, B	$A + B \rightarrow B$
`negq` A	$-A \rightarrow A$
`subq` A, B	$B - A \rightarrow B$
`callq` L	Pushes the return address and jumps to label L
`callq` `*`A	Calls the function at the address A.
`retq`	Pops the return address and jumps to it
`popq` A	$*\texttt{rsp} \rightarrow A; \texttt{rsp} + 8 \rightarrow \texttt{rsp}$
`pushq` A	$\texttt{rsp} - 8 \rightarrow \texttt{rsp}; A \rightarrow *\texttt{rsp}$
`leaq` A,B	$A \rightarrow B$ (B must be a register)
`cmpq` A, B	compare A and B and set the flag register (B must not be an immediate)
`je` L	Jump to label L if the flag register matches the
`jl` L	condition code of the instruction, otherwise go to the
`jle` L	next instructions. The condition codes are `e` for
`jg` L	"equal", `l` for "less", `le` for "less or equal", `g` for
`jge` L	"greater", and `ge` for "greater or equal".
`jmp` L	Jump to label L
`movq` A, B	$A \rightarrow B$
`movzbq` A, B	$A \rightarrow B$, where A is a single-byte register (e.g., `al` or `cl`), B is a 8-byte register, and the extra bytes of B are set to zero.
`notq` A	$\sim A \rightarrow A$ (bitwise complement)
`orq` A, B	$A \mid B \rightarrow B$ (bitwise-or)
`andq` A, B	$A \& B \rightarrow B$ (bitwise-and)
`salq` A, B	$B \ll A \rightarrow B$ (arithmetic shift left, where A is a constant)
`sarq` A, B	$B \gg A \rightarrow B$ (arithmetic shift right, where A is a constant)
`sete` A `setl` A `setle` A `setg` A `setge` A	If the flag matches the condition code, then $1 \rightarrow A$, else $0 \rightarrow A$. Refer to `je` above for the description of the condition codes. A must be a single byte register (e.g., `al` or `cl`).

Table 12.1: Quick-reference for the x86 instructions used in this book.

Index

Bibliography

Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2nd edition, 1996. ISBN 0262011530.

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986. ISBN 0-201-10088-6.

Hussein Al-Omari and Khair Eddin Sabri. New graph coloring algorithms. *Journal of Mathematics and Statistics*, 2(4), 2006.

Frances E. Allen. Control flow analysis. In *Proceedings of a symposium on Compiler optimization*, pages 1–19, 1970.

Andrew W. Appel. Runtime tags aren't necessary. *LISP and Symbolic Computation*, 2(2):153–162, 1989. ISSN 0892-4635. doi: 10.1007/BF01811537. URL `http://dx.doi.org/10.1007/BF01811537`.

Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, 2003. ISBN 052182060X.

J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Report on the algorithmic language algol 60. *Commun. ACM*, 3(5):299–314, May 1960. ISSN 0001-0782. doi: 10.1145/367236.367262. URL `http://doi.acm.org/10.1145/367236.367262`.

J. Baker, A. Cunei, T. Kalibera, F. Pizlo, and J. Vitek. Accurate garbage collection in uncooperative environments revisited. *Concurr. Comput. : Pract. Exper.*, 21(12):1572–1606, August 2009. ISSN 1532-0626. doi: 10.1002/cpe.v21:12. URL `http://dx.doi.org/10.1002/cpe.v21:12`.

V. K. Balakrishnan. *Introductory Discrete Mathematics*. Dover Publications, Incorporated, 1996. ISBN 0486691152.

Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Myths and realities: The performance impact of garbage collection. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '04/Performance '04, pages 25–36, New York, NY, USA, 2004. ACM. ISBN 1-58113-873-3. doi: 10.1145/1005686.1005693. URL `http://doi.acm.org/10.1145/1005686.1005693`.

Daniel Brélaz. New methods to color the vertices of a graph. *Commun. ACM*, 22(4):251–256, 1979. ISSN 0001-0782.

Randal E. Bryant and David R. O'Hallaron. *x86-64 Machine-Level Programming*. Carnegie Mellon University, September 2005.

Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Addison-Wesley Publishing Company, USA, 2nd edition, 2010. ISBN 0136108040, 9780136108047.

Luca Cardelli. The functional abstract machine. Technical Report TR-107, AT&T Bell Laboratories, 1983.

Luca Cardelli. Compiling a functional language. In *ACM Symposium on LISP and Functional Programming*, LFP '84, pages 208–217. ACM, 1984.

Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–523, 1985. ISSN 0360-0300.

C. J. Cheney. A nonrecursive list compacting algoirthm. *Communications of the ACM*, 13(11), 1970.

George E. Collins. A method for overlapping and erasure of lists. *Commun. ACM*, 3(12):655–657, December 1960. ISSN 0001-0782. doi: 10.1145/367487.367501. URL `https://doi.org/10.1145/367487.367501`.

Keith Cooper and Linda Torczon. *Engineering a Compiler*. Morgan Kaufmann, 2nd edition, 2011.

Keith D. Cooper and L. Taylor Simpson. Live range splitting in a graph coloring register allocator. In *Compiler Construction*, 1998.

Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001. ISBN 0070131511.

Cody Cutler and Robert Morris. Reducing pause times with clustered collection. In *Proceedings of the 2015 International Symposium on Memory Management*, ISMM '15, pages 131–142, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3589-8. doi: 10.1145/2754169.2754184. URL `http://doi.acm.org/10.1145/2754169.2754184`.

Olivier Danvy. Three steps for the CPS transformation. Technical Report CIS-92-02, Kansas State University, December 1991.

David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. Garbage-first garbage collection. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04, pages 37–48, New York, NY, USA, 2004. ACM. ISBN 1-58113-945-4. doi: 10.1145/1029873.1029879. URL `http://doi.acm.org/10.1145/1029873.1029879`.

E. W. Dijkstra. Why numbering should start at zero. Technical Report EWD831, University of Texas at Austin, 1982.

Amer Diwan, Eliot Moss, and Richard Hudson. Compiler support for garbage collection in a statically typed language. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, pages 273–282, New York, NY, USA, 1992. ACM. ISBN 0-89791-475-9. doi: 10.1145/143095.143140. URL `http://doi.acm.org/10.1145/143095.143140`.

R. Kent Dybvig. *The SCHEME Programming Language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987. ISBN 0-13-791864-X.

R. Kent Dybvig. The development of chez scheme. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '06, pages 1–12, New York, NY, USA, 2006. ACM. ISBN 1-59593-309-3. doi: 10.1145/1159803.1159805. URL `http://doi.acm.org/10.1145/1159803.1159805`.

R. Kent Dybvig and Andrew Keep. P523 compiler assignments. Technical report, Indiana University, 2010.

Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD-machine and the lambda-calculus. pages 193–217, 1986.

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to Design Programs: An Introduction to Programming and Computing*. MIT Press, Cambridge, MA, USA, 2001. ISBN 0-262-06218-6.

Matthias Felleisen, M.D. Barski Conrad, David Van Horn, and Eight Students of Northeastern University. *Realm of Racket: Learn to Program, One Game at a Time!* No Starch Press, San Francisco, CA, USA, 2013. ISBN 1593274912, 9781593274917.

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Conference on Programming Language Design and Implementation*, PLDI, pages 502–514, June 1993.

Matthew Flatt and PLT. The Racket reference 6.0. Technical report, PLT Inc., 2014. `http://docs.racket-lang.org/reference/index.html`.

Matthew Flatt, Robert Bruce Findler, and PLT. The racket guide. Technical Report 6.0, PLT Inc., 2014.

Daniel P. Friedman and Matthias Felleisen. *The Little Schemer (4th Ed.)*. MIT Press, Cambridge, MA, USA, 1996. ISBN 0-262-56099-2.

Daniel P. Friedman and David S. Wise. Cons should not evaluate its arguments. Technical Report TR44, Indiana University, 1976.

Ben Gamari and Laura Dietz. Alligator collector: A latency-optimized garbage collector for functional programming languages. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2020, pages 87–99, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450375665. doi: 10.1145/3381898.3397214. URL `https://doi.org/10.1145/3381898.3397214`.

Assefaw Hadish Gebremedhin. *Parallel Graph Coloring*. PhD thesis, University of Bergen, 1999.

Abdulaziz Ghuloum. An incremental approach to compiler construction. In *Scheme and Functional Programming Workshop*, 2006.

Benjamin Goldberg. Tag-free garbage collection for strongly typed programming languages. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91,

pages 165–176, New York, NY, USA, 1991. ACM. ISBN 0-89791-428-7. doi: 10.1145/113445.113460. URL `http://doi.acm.org/10.1145/113445.113460`.

Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 130–141. ACM Press, 1995. ISBN 0-89791-692-1.

Fergus Henderson. Accurate garbage collection in an uncooperative environment. In *Proceedings of the 3rd International Symposium on Memory Management*, ISMM '02, pages 150–156, New York, NY, USA, 2002. ACM. ISBN 1-58113-539-4. doi: 10.1145/512429.512449. URL `http://doi.acm.org/10.1145/512429.512449`.

Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B, 3C and 3D*, December 2015.

Nicholas Jacek and J. Eliot B. Moss. Learning when to garbage collect with random forests. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2019, pages 53–63, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367226. doi: 10.1145/3315573.3329983. URL `https://doi.org/10.1145/3315573.3329983`.

Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993. ISBN 0-13-020249-5.

Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, Inc., New York, NY, USA, 1996. ISBN 0-471-94148-4.

Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 1st edition, 2011. ISBN 1420082795, 9781420082791.

Andrew W. Keep. *A Nanopass Framework for Commercial Compiler Development*. PhD thesis, Indiana University, December 2012.

R. Kelsey, W. Clinger, and J. Rees (eds.). Revised^5 report on the algorithmic language scheme. *Higher-Order and Symbolic Computation*, 11(1), August 1998.

Brian W. Kernighan and Dennis M. Ritchie. *The C programming language*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1988. ISBN 0-13-110362-8.

Donald E. Knuth. Backus normal form vs. backus naur form. *Commun. ACM*, 7(12):735–736, December 1964. ISSN 0001-0782. doi: 10.1145/355588.365140. URL `http://doi.acm.org/10.1145/355588.365140`.

Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 151–161, New York, NY, USA, 1986. ACM. ISBN 0-89791-200-4.

Xavier Leroy. Unboxed objects and polymorphic typing. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 177–188, New York, NY, USA, 1992. ACM Press. ISBN 0-89791-453-8.

Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM*, 26(6):419–429, June 1983. ISSN 0001-0782. doi: 10.1145/358141.358147. URL `http://doi.acm.org/10.1145/358141.358147`.

Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. *System V Application Binary Interface, AMD64 Architecture Processor Supplement*, October 2013.

John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, 1960. ISSN 0001-0782.

E.F. Moore. The shortest path through a maze. In *Proceedings of an International Symposium on the Theory of Switching*, April 1959.

Erik Österlund and Welf Löwe. Block-free concurrent gc: Stack scanning and copying. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2016, pages 1–12, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450343176. doi: 10.1145/2926697.2926701. URL `https://doi.org/10.1145/2926697.2926701`.

Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM '72: Proceedings of the ACM Annual Conference*, pages 717–740. ACM Press, 1972.

Kenneth H. Rosen. *Discrete Mathematics and Its Applications*. McGraw-Hill Higher Education, 2002. ISBN 0072474777.

Dipanwita Sarkar, Oscar Waddell, and R. Kent Dybvig. A nanopass infrastructure for compiler education. In *ICFP '04: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, pages 201–212. ACM Press, 2004. ISBN 1-58113-905-5.

Rifat Shahriyar, Stephen M. Blackburn, Xi Yang, and Kathryn M. McKinley. Taking off the gloves with reference counting immix. In *OOPSLA '13: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, oct 2013. doi: http://dx.doi.org/10.1145/2509136.2509527.

Zhong Shao. Flexible representation analysis. In *ICFP '97: Proceedings of the 2nd ACM SIGPLAN international conference on Functional programming*, pages 85–98, New York, NY, USA, 1997. ACM Press. ISBN 0-89791-918-1.

Jonathan Shidal, Ari J. Spilo, Paul T. Scheid, Ron K. Cytron, and Krishna M. Kavi. Recycling trash in cache. In *Proceedings of the 2015 International Symposium on Memory Management*, ISMM '15, pages 118–130, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3589-8. doi: 10.1145/2754169.2754183. URL `http://doi.acm.org/10.1145/2754169.2754183`.

Fridtjof Siebert. *Compiler Construction: 10th International Conference, CC 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2–6, 2001 Proceedings*, chapter Constant-Time Root Scanning for Deterministic Garbage Collection, pages 304–318. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001. ISBN 978-3-540-45306-2. doi: 10.1007/3-540-45306-7_21. URL `http://dx.doi.org/10.1007/3-540-45306-7_21`.

Jeremy G. Siek and Bor-Yuh Evan Chang. A problem course in compilation: From python to x86 assembly. Technical report, Univesity of Colorado, 2012.

Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, pages 81–92, September 2006.

Michael Sperber, R. KENT DYBVIG, MATTHEW FLATT, ANTON VAN STRAATEN, ROBBY FINDLER, and JACOB MATTHEWS. Revised^6 report on the algorithmic language scheme. *Journal of Functional Programming*, 19:1–301, 8 2009. ISSN 1469-7653. doi: 10.1017/S0956796809990074. URL `http://journals.cambridge.org/article_S0956796809990074`.

Guy L. Steele, Jr. Data representations in pdp-10 maclisp. AI Memo 420, MIT Artificial Intelligence Lab, September 1977.

Gerald Jay Sussman and Guy L. Steele Jr. Scheme: an interpreter for extended lambda calculus. Technical Report AI Memo No. 349, MIT, December 1975.

Gil Tene, Balaji Iyengar, and Michael Wolf. C4: the continuously concurrent compacting collector. In *Proceedings of the international symposium on Memory management*, ISMM '11, pages 79–88, New York, NY, USA, 2011. ACM. doi: http://doi.acm.org/10.1145/1993478.1993491.

David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, SDE 1, pages 157–167, New York, NY, USA, 1984. ACM. ISBN 0-89791-131-8. doi: 10.1145/800020.808261. URL `http://doi.acm.org/10.1145/800020.808261`.

Oscar Waddell and R. Kent Dybvig. Fast and effective procedure inlining. In *Proceedings of the 4th International Symposium on Static Analysis*, SAS '97, pages 35–52, London, UK, 1997. Springer-Verlag.

Paul Wilson. Uniprocessor garbage collection techniques. In Yves Bekkers and Jacques Cohen, editors, *Memory Management*, volume 637 of *Lecture Notes in Computer Science*, pages 1–42. Springer Berlin / Heidelberg, 1992. URL `http://dx.doi.org/10.1007/BFb0017182`. 10.1007/BFb0017182.