Essentials of Compilation
An Incremental Approach

Jeremy G. Siek, Ryan R. Newton
Indiana University

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

December 19, 2018

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 had conducted research on lazy evaluation [**?**] in the context of Lisp [McCarthy, 1960] and then studied continuations [**?**] and macros [**?**] 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!

One of Jeremy's classmates, 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, then in each subsequent stage they add a feature to the input language 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 absolute 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. We do not recommend this book for students who want to dabble 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 other 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 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

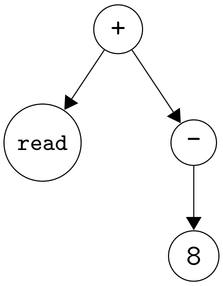
Preliminaries

In this chapter, we review the basic tools that are needed for implementing a compiler. We use abstract syntax trees (ASTs), which refer to data structures in the compilers memory, rather than programs as they are stored on disk, in *concrete syntax*. ASTs can be represented in many different ways, depending on the programming language used to write the compiler. Because this book uses Racket (`http://racket-lang.org`), a descendant of Lisp, we use S-expressions to represent programs (Section 1.1). We use grammars to defined 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 S-expressions

The primary data structure that is commonly used for representing programs is the *abstract syntax tree* (AST). When considering some part of a program, a compiler needs to ask what kind of part it is and what sub-parts it has. For example, the program on the left, represented by an S-expression, corresponds to the AST on the right.

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



(1.1)

We shall use the standard terminology for trees: 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.

Recall that an *symbolic expression* (S-expression) is either

1. an atom, or
2. a pair of two S-expressions, written $(e_1 \texttt{.} e_2)$, where e_1 and e_2 are each an S-expression.

An *atom* can be a symbol, such as `‘hello`, a number, the null value `’()`, etc. We can create an S-expression in Racket simply by writing a backquote (called a quasi-quote in Racket). followed by the textual representation of the S-expression. It is quite common to use S-expressions to represent a list, such as a, b, c in the following way:

```
‘(a . (b . (c . ())))
```

Each element of the list is in the first slot of a pair, and the second slot is either the rest of the list or the null value, to mark the end of the list. Such lists are so common that Racket provides special notation for them that removes the need for the periods and so many parenthesis:

```
‘(a b c)
```

For another example, an S-expression to represent the AST (1.1) is created by the following Racket expression:

```
‘(+ (read) (- 8))
```

When using S-expressions to represent ASTs, the convention is to represent each AST node as a list and to put the operation symbol at the front of the list. The rest of the list contains the children. So in the above case, the root

AST node has operation `'+` and its two children are `'(read)` and `'(- 8)`, just as in the diagram (1.1).

To build larger S-expressions one often needs to splice together several smaller S-expressions. Racket provides the comma operator to splice an S-expression into a larger one. For example, instead of creating the S-expression for AST (1.1) all at once, we could have first created an S-expression for AST (1.5) and then spliced that into the addition S-expression.

```
(define ast1.4 `(- 8))
(define ast1.1 `(+ (read) ,ast1.4))
```

In general, the Racket expression that follows the comma (splice) can be any expression that computes an S-expression.

When deciding how to compile program (1.1), we need to know that the operation associated with the root node is addition and that it has two children: `read` and a negation. The AST data structure directly supports these queries, as we shall see in Section 1.3, and hence is a good choice for use in compilers. In this book, we will often write down the S-expression representation of a program even when we really have in mind the AST because the S-expression 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. 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, of integers and arithmetic operations. The first rule says that any integer is an expression, *exp*, in the language:

$$\mathit{exp} ::= \mathit{int} \tag{1.2}$$

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, however, we do not define *int* because the reader already knows what an integer is. Further, 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 $2^{63}-1$. However, we restrict this range further to match the Racket `fixnum` datatype, which allows 63-bit integers on a 64-bit machine.

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

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

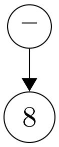
The third rule says that, given an *exp* node, you can build another *exp* node by negating it.

$$exp ::= (\texttt{-}\ exp) \tag{1.4}$$

Symbols such as `-` in typewriter font 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), `8` is an *exp*, then by rule (1.4), the following AST is an *exp*.

`(- 8)`



(1.5)

The following grammar rule defines addition expressions:

$$exp ::= (\texttt{+}\ exp\ exp) \tag{1.6}$$

Now we can see that the AST (1.1) is an *exp* in R_0. We know that `(read)` is an *exp* by rule (1.3) and we have shown that `(- 8)` is an *exp*, so we can apply rule (1.6) to show that `(+ (read) (- 8))` 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 AST `(- (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, we implicitly mean for the language to be the smallest set of programs that are justified by the rules. That is, the language only includes those programs that the rules allow.

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

$$R_0 ::= (\texttt{program}\ exp)$$

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

Figure 1.1: The syntax of R_0, a language of integer arithmetic.

The `read-program` function provided in `utilities.rkt` reads programs in from a file (the sequence of characters in the concrete syntax of Racket) and parses them into the abstract syntax tree. The concrete syntax does not include a `program` form; that is added by the `read-program` function as it creates the AST. See the description of `read-program` in Appendix 12.2 for more details.

It is common to have many rules with the same left-hand side, such as *exp* in the grammar for R_0, so there is a vertical bar notation for gathering several rules, as shown in Figure 1.1. Each clause between a vertical bar is called an *alternative*.

1.3 Pattern Matching

As mentioned above, one of the operations that a compiler needs to perform on an AST is to access the children of a node. Racket provides the `match` form to access the parts of an S-expression. Consider the following example and the output on the right.

```
(match ast1.1
  [`(,op ,child1 ,child2)
    (print op) (newline)              '+
    (print child1) (newline)          '(read)
    (print child2)])                  '(- 8)
```

The `match` form takes AST (1.1) and binds its parts to the three variables `op`, `child1`, and `child2`. In general, a match clause consists of a *pattern* and a *body*. The pattern is a quoted S-expression that may contain pattern-variables (each one preceded by a comma). The pattern is not the same thing as a quasiquote expression used to *construct* ASTs, however, the similarity is intentional: constructing and deconstructing ASTs uses similar syntax. While the pattern uses a restricted syntax, the body of the match clause may contain any Racket code whatsoever.

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 S-expression. The body of the first clause that matches is executed. The output of `leaf?` for several S-expressions is shown on the right. In the below `match`, we see another form of pattern: the `(? fixnum?)` applies the predicate `fixnum?` to the input S-expression to see if it is a machine-representable integer.

```
(define (leaf? arith)
  (match arith
    [(? fixnum?) #t]
    [`(read) #t]
    [`(- ,c1) #f]
    [`(+ ,c1 ,c2) #f]))

(leaf? `(read))                       #t
(leaf? `(- 8))                        #f
(leaf? `(+ (read) (- 8)))             #f
```

1.4 Recursion

Programs are inherently recursive in that an R_0 expression (*exp*) is made up of smaller expressions. Thus, the natural way to process an entire program is with a recursive function. As a first example of such a function, we define `exp?` below, which takes an arbitrary S-expression, `sexp`, and determines whether or not `sexp` is an R_0 expression. Note that each match clause corresponds to one grammar rule the body of each clause makes a recursive call for each child node. This pattern of recursive function is so common that it has a name, *structural recursion*. In general, when a recursive function is defined using a sequence of match clauses that correspond to a grammar, and each clause body makes a recursive call on each child node, then we say the function is defined by structural recursion. Below we also define a second function, named `R0?`, determines whether an S-expression is an R_0 program.

```
(define (exp? sexp)
  (match sexp
    [(? fixnum?) #t]
    [`(read) #t]
    [`(- ,e) (exp? e)]
    [`(+ ,e1 ,e2)
      (and (exp? e1) (exp? e2))]
    [else #f]))

(define (R0? sexp)
  (match sexp
    [`(program ,e) (exp? e)]
    [else #f]))

(R0? `(program (+ (read) (- 8))))              #t
(R0? `(program (- (read) (+ 8))))              #f
```

Indeed, the structural recursion follows the grammar itself. We can generally expect to write a recursive function to handle each non-terminal in the grammar.[1]

You may be tempted to write the program with just one function, like this:

```
(define (R0? sexp)
  (match sexp
    [(? fixnum?) #t]
    [`(read) #t]
    [`(- ,e) (R0? e)]
    [`(+ ,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 instance, the above function is subtly wrong: `(R0? `(program (program 3)))` will return true, when it should return false.

[1] 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 (interp-exp e)
  (match e
    [(? fixnum?) e]
    [`(read)
     (let ([r (read)])
       (cond [(fixnum? r) r]
             [else (error 'interp-R0 "input␣not␣an␣integer" r)]))]
    [`(- ,e1)    (fx- 0 (interp-exp e1))]
    [`(+ ,e1 ,e2) (fx+ (interp-exp e1) (interp-exp e2))]
    ))

(define (interp-R0 p)
  (match p
    [`(program ,e) (interp-exp e)]))
```

Figure 1.2: Interpreter for the R_0 language.

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 to define the meaning of each language that we consider, following Reynold's advice in this regard [Reynolds, 1972]. Here we warm up by writing an interpreter for the R_0 language, which serves as a second example of structural recursion. The `interp-R0` function is defined in Figure 1.2. The body of the function is a match on the input program `p` and then 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 simply adds two integers.

```
(+ 10 32)
```

The result is `42`, as you might have expected. Here we have written the program in concrete syntax, whereas the parsed abstract syntax would be the slightly different: `(program (+ 10 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 $R0$ language does not support arbitrarily-large integers, but only 63-bit integers, so we interpret the arithmetic operations of $R0$ using fixnum arithmetic. What happens when we run the following program?

```
(define large 999999999999999999)
(interp-R0 `(program (+ (+ (+ ,large ,large) (+ ,large ,large))
                        (+ (+ ,large ,large) (+ ,large ,large)))))
```

It produces an error:

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

We shall use the convention that if the interpreter for a language produces an error when run on a program, then the meaning of the program is unspecified. The compiler for the language is under no obligation for such a program; it can produce an executable that does anything.
Moving on, the `read` operation prompts the user of the program for an integer. If we interpret the AST (1.1) and give it the input `50`

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

we get the answer to life, the universe, and everything:

```
42
```

We include the `read` operation in R_0 so a clever student cannot implement a compiler for R_0 simply by running the interpreter at compilation time to obtain the output and then generating the trivial code to return the output. (A clever student did this in a previous version of the 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. 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 interpreting P_1 and P_2 on their respective interpreters with input i should yield the same output o.

$$\begin{tikzcd}
P_1 \arrow[r, "\text{compile}"] \arrow[dr, "\text{interp-}\mathcal{L}_1(i)"'] & P_2 \arrow[d, "\text{interp-}\mathcal{L}_2(i)"] \\
 & o
\end{tikzcd} \tag{1.7}$$

In the next section we see our first example of a compiler, which is another example of structural recursion.

1.6 Example Compiler: a Partial Evaluator

In this section we consider a compiler that translates R_0 programs into R_0 programs that are more efficient, that is, this compiler is an optimizer. Our optimizer will accomplish this by trying to eagerly compute the parts of the program that do not depend on any inputs. For example, given the following program

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

our compiler will translate it into the program

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

Figure 1.3 gives the code for a simple partial evaluator for the R_0 language. The output of the partial evaluator is an R_0 program, which we build up using a combination of quasiquotes and commas. (Though no quasiquote is necessary for integers.) In Figure 1.3, the normal structural recursion is captured in the main `pe-arith` function whereas the code for partially evaluating negation and addition 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 nodes.

Our code for `pe-neg` and `pe-add` implements the simple idea of checking whether their arguments are integers and if they are, to go ahead and perform the arithmetic. Otherwise, we use quasiquote to create an AST node for the appropriate operation (either negation or addition) and use comma to splice in the child nodes.

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 program. 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 `assert` function is defined in Appendix 12.2.

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

(test-pe ‘(+ (read) (- (+ 5 3))))
(test-pe ‘(+ 1 (+ (read) 1)))
(test-pe ‘(- (+ (read) (- 5))))
```

```
(define (pe-neg r)
  (cond [(fixnum? r) (fx- 0 r)]
        [else `(- ,r)]))

(define (pe-add r1 r2)
  (cond [(and (fixnum? r1) (fixnum? r2)) (fx+ r1 r2)]
        [else `(+ ,r1 ,r2)]))

(define (pe-arith e)
  (match e
    [(? fixnum?) e]
    [`(read) `(read)]
    [`(- ,e1)
      (pe-neg (pe-arith e1))]
    [`(+ ,e1 ,e2)
      (pe-add (pe-arith e1) (pe-arith e2))]))
```

Figure 1.3: A partial evaluator for R_0 expressions.

Exercise 1. We challenge the reader to improve on the simple partial evaluator in Figure 1.3 by replacing the `pe-neg` and `pe-add` helper functions with functions that know more about arithmetic. For example, your partial evaluator should translate

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

into

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

To accomplish this, we recommend that your partial evaluator produce output that takes the form of the *residual* non-terminal in the following grammar.

exp ::= `(read)` | `(- (read))` | `(+` *exp exp*`)`
residual ::= *int* | `(+` *int exp*`)` | *exp*

2

Integers and Variables

This chapter concerns the challenge of compiling a 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 quite large, so we only discuss 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 2.10). We hope to give enough hints that the well-prepared reader can implement a compiler from R_1 to x86 while at the same time leaving room for some fun and creativity.

2.1 The R_1 Language

The R_1 language extends the R_0 language (Figure 1.1) with variable definitions. The syntax of the R_1 language is defined by the grammar in Figure 2.1. 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 R_1 language includes the `program` construct to mark the top of the program, which is helpful in parts of the compiler. The *info* field of the `program` construct contain an association list that is used to communicating auxiliary data from one step of the compiler to the next.

The R_1 language is rich enough to exhibit several compilation techniques but simple enough so that the reader, together with couple friends, can

$$
\begin{array}{rcl}
\mathit{exp} & ::= & \mathit{int} \mid (\texttt{read}) \mid (\texttt{-}\ \mathit{exp}) \mid (\texttt{+}\ \mathit{exp}\ \mathit{exp}) \\
 & \mid & \mathit{var} \mid (\texttt{let}\ ([\mathit{var}\ \mathit{exp}])\ \mathit{exp}) \\
R_1 & ::= & (\texttt{program}\ \mathit{info}\ \mathit{exp})
\end{array}
$$

Figure 2.1: The syntax of R_1, a language of integers and variables.

implement a compiler for it in a week or two of part-time work. 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.

Let us dive into the description of the R_1 language. The `let` construct defines a variable for use within its body and initializes the variable with the value of an expression. So the following program initializes `x` to `32` and then evaluates the body `(+ 10 x)`, producing `42`.

```
(program ()
  (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?

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

For the purposes of showing 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.

```
(program ()
  (let ([x$_1$ 32]) (+ (let ([x$_2$ 10]) x$_2$) x$_1$)))
```

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` (and not `-42`).

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

Figure 2.2 shows the 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 will need a way to communicate the initializing value of a variable to all the uses of a variable. To accomplish this, we maintain a mapping from variables to values, which is traditionally called an *environment*. For

```
(define (interp-exp env)
  (lambda (e)
    (match e
      [(? fixnum?) e]
      [`(read)
       (define r (read))
       (cond [(fixnum? r) r]
             [else (error 'interp-R1 "expected␣an␣integer" r)])]
      [`(- ,e)
       (define v ((interp-exp env) e))
       (fx- 0 v)]
      [`(+ ,e1 ,e2)
       (define v1 ((interp-exp env) e1))
       (define v2 ((interp-exp env) e2))
       (fx+ v1 v2)]
      [(? symbol?) (lookup e env)]
      [`(let ([,x ,e]) ,body)
       (define new-env (cons (cons x ((interp-exp env) e)) env))
       ((interp-exp new-env) body)]
      )))

   (define (interp-R1 env)
     (lambda (p)
       (match p
         [`(program ,info ,e) ((interp-exp '()) e)])))
```

Figure 2.2: Interpreter for the R_1 language.

simplicity, here we use an association list to represent the environment. 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 `lookup` function (Appendix 12.2). When the interpreter encounters a `let`, it evaluates the initializing expression, extends the environment with the result bound to the variable, then evaluates the body of the `let`.

The goal for this chapter is to implement a compiler that translates any program P_1 in the R_1 language into an x86 assembly program P_2 such that P_2 exhibits the same behavior on an x86 computer as the R_1 program running in a Racket implementation. That is, they both output the same integer n.

$$\begin{tikzcd}
P_1 \arrow[r, "\text{compile}"] \arrow[dr, swap, "\text{interp-}R_1"] & P_2 \arrow[d, "\text{interp-x86}"] \\
 & n
\end{tikzcd}$$

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

2.2 The x86 Assembly Language

An x86 program is a sequence of instructions. The program is stored in the computer's memory and the *program counter* 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. Each instruction may refer to integer constants (called *immediate values*), variables called *registers*, and instructions may load and store values into memory. For our purposes, we can think of the computer's memory as a mapping of 64-bit addresses to 64-bit values[1]. Figure 2.3 defines the syntax for the subset of the x86 assembly language needed for this chapter. 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. Also, Appendix 12.3 includes a quick-reference of all the x86 instructions used in this book and a short explanation of what they do.

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 offsets the address by n bytes (8 bits). The address is then 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 in d.

[1] 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.

$$
\begin{array}{lcl}
\mathit{reg} & ::= & \texttt{rsp} \mid \texttt{rbp} \mid \texttt{rax} \mid \texttt{rbx} \mid \texttt{rcx} \mid \texttt{rdx} \mid \texttt{rsi} \mid \texttt{rdi} \mid \\
 & & \texttt{r8} \mid \texttt{r9} \mid \texttt{r10} \mid \texttt{r11} \mid \texttt{r12} \mid \texttt{r13} \mid \texttt{r14} \mid \texttt{r15} \\
\mathit{arg} & ::= & \texttt{\$}\mathit{int} \mid \%\mathit{reg} \mid \mathit{int}(\%\mathit{reg}) \\
\mathit{instr} & ::= & \texttt{addq}\ \mathit{arg}, \mathit{arg} \mid \texttt{subq}\ \mathit{arg}, \mathit{arg} \mid \texttt{negq}\ \mathit{arg} \mid \texttt{movq}\ \mathit{arg}, \mathit{arg} \mid \\
 & & \texttt{callq}\ \mathit{label} \mid \texttt{pushq}\ \mathit{arg} \mid \texttt{popq}\ \mathit{arg} \mid \texttt{retq} \mid \mathit{label}\texttt{:}\ \mathit{instr} \\
\mathit{prog} & ::= & \texttt{.globl main} \\
 & & \texttt{main:}\ \mathit{instr}^{+}
\end{array}
$$

Figure 2.3: A subset of the x86 assembly language (AT&T syntax).

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

Figure 2.4: An x86 program equivalent to (`+` 10 32).

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.

Figure 2.4 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 the program was successful, 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 2.5 lists an x86 program that is equivalent to (`+` 52 (`-` 10)).

```
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 2.5: An x86 program 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 Figure 2.6. 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. The frame size is required to be a multiple of 16 bytes. In the context of a procedure call, the *return address* is the next instruction on the caller side that comes after the call instruction. During a function call, the return address is pushed onto the stack. The register `rbp` is the *base pointer* which serves two purposes: 1) it saves the location of the stack pointer for the calling procedure and 2) it is used to access variables associated with the current procedure. The base pointer of the calling procedure 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 2.5, the first three instructions are the typical *prelude* for a procedure. The instruction `pushq %rbp` saves the base pointer for the procedure that called the current one onto the stack and subtracts 8 from the stack pointer. The second instruction `movq %rsp, %rbp` changes the base pointer to the top of the stack. The instruction `subq`

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

Figure 2.6: Memory layout of a frame.

`$16, %rsp` moves the stack pointer down to make enough room for storing variables. This program just needs one variable (8 bytes) but because the frame size is required to be a multiple of 16 bytes, it rounds to 16 bytes.

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 `movq $52, %rax` places 52 in the register `rax` and `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 finale of a procedure. The first two are necessary to get the state of the machine back to where it was at the beginning of the procedure. The `addq $16, %rsp` instruction 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 will need a convenient representation for manipulating x86 programs, so we define an abstract syntax for x86 in Figure 2.7. 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 2.3) is that it does nto allow labelled instructions to appear anywhere, but instead organizes instructions into groups called *blocks* and a label is associated with every block, which is why the `program` form includes an association list mapping labels to blocks. The reason for this organization becomes apparent in Chapter 4.

$$
\begin{array}{lcl}
\mathit{register} & ::= & \texttt{rsp} \mid \texttt{rbp} \mid \texttt{rax} \mid \texttt{rbx} \mid \texttt{rcx} \mid \texttt{rdx} \mid \texttt{rsi} \mid \texttt{rdi} \mid \\
 & & \texttt{r8} \mid \texttt{r9} \mid \texttt{r10} \mid \texttt{r11} \mid \texttt{r12} \mid \texttt{r13} \mid \texttt{r14} \mid \texttt{r15} \\
\mathit{arg} & ::= & (\texttt{int}\ \mathit{int}) \mid (\texttt{reg}\ \mathit{register}) \mid (\texttt{deref}\ \mathit{register}\ \mathit{int}) \\
\mathit{instr} & ::= & (\texttt{addq}\ \mathit{arg}\ \mathit{arg}) \mid (\texttt{subq}\ \mathit{arg}\ \mathit{arg}) \mid (\texttt{movq}\ \mathit{arg}\ \mathit{arg}) \mid (\texttt{retq}) \\
 & \mid & (\texttt{negq}\ \mathit{arg}) \mid (\texttt{callq}\ \mathit{label}) \mid (\texttt{pushq}\ \mathit{arg}) \mid (\texttt{popq}\ \mathit{arg}) \\
\mathit{block} & ::= & (\texttt{block}\ \mathit{info}\ \mathit{instr}^{+}) \\
x86_0 & ::= & (\texttt{program}\ \mathit{info}\ ((\mathit{label}\ .\ \mathit{block})^{+}))
\end{array}
$$

Figure 2.7: Abstract syntax for $x86_0$ assembly.

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 them. What are the differences between R_1 and x86 assembly? Here we list 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 to an R_1 operator can be any expression, whereas x86 instructions restrict their arguments to be *simple expressions* like integers, registers, and memory locations. (All the other kinds are called *complex expressions*.)

(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 it 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, because step traverses (passes over) the AST of the program. We begin by giving a sketch about how we might implement each pass, and give them names. We shall 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 the compiler, we shall write one function, typically a structural recursive function, per pass.

Pass `select-instructions` To handle the difference between R_1 operations and x86 instructions we shall 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 `opera*`) is a simple expression, we shall introduce temporary variables to hold the results of subexpressions.

Pass `explicate-control` To make the execution order of the program explicit, we shall convert from the abstract syntax tree representation into a graph representation in which each node contains a sequence of actions and the edges say where to go after the sequence is complete.

Pass `assign-homes` To handle the difference between the variables in R_1 versus the registers and stack location in x86, we shall come up with an assignment of each variable to its *home*, that is, 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 a challenging one to answer because it is difficult to know ahead of time which orders will be better (easier to implement, produce more efficient code, etc.) so often some trial-and-error is involved. Nevertheless, we can try to plan ahead and make educated choices regarding the orderings.

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.

Next we shall consider the ordering of the explicate-control pass and select-instructions. It is clear that explicate-control must come first because the control-flow graph that it generates is needed when determining where to place the x86 label and jump instructions. Regarding the ordering of explicate-control with respect to uniquify, it is important to apply uniquify first because in explicate-control we change all the let-bound variables to become local variables whose scope is the entire program. With respect to remove-complex-opera*, it perhaps does not matter very much, but it works well to place explicate-control after removing complex subexpressions.

The assign-homes pass should come after remove-complex-opera* and explicate-control. The remove-complex-opera* pass generates temporary variables, which also need to be assigned homes. The explicate-control pass deletes branches that will never be executed, which can remove variables. Thus it is good to place explicate-control prior to assign-homes so that there are fewer variables that need to be assigned homes. This is important because the assign-homes pass has the highest time complexity.

Last, we need to decide on the ordering of select-instructions and assign-homes. These two issues are intertwined, creating a bit of 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. For example, one can 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, followed by a third pass, named patch-instructions, that uses a reserved register (rax) to patch-up outstanding problems regarding instructions with too many memory accesses.

Figure 2.8 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* are programs that are still in the R_1 language, but the output of the pass explicate-control is in a different language that is designed to make the order of evaluation explicit in its syntax, which we introduce in the next section. Also, there are two passes of lesser importance in Figure 2.8 that

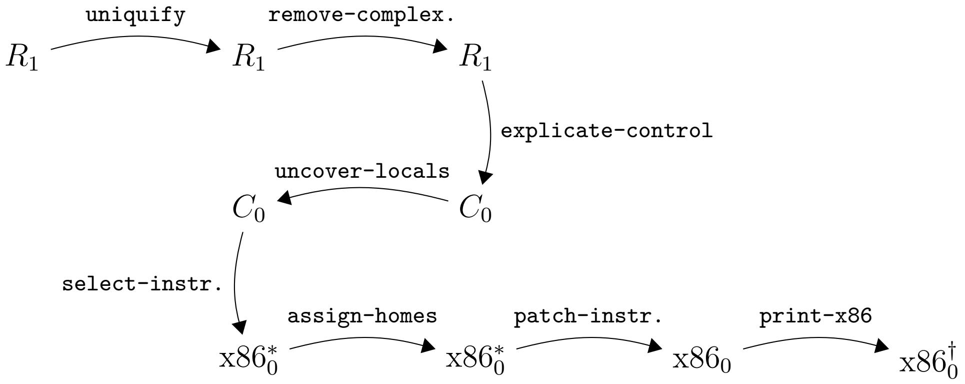


Figure 2.8: Overview of the passes for compiling R_1.

we have not yet talked about, `uncover-locals` and `print-x86`. We shall discuss them later in this Chapter.

2.3.1 The C_0 Intermediate Language

It so happens that the output of `explicate-control` is vaguely similar to the *C* language [Kernighan and Ritchie, 1988], so we name it C_0. The syntax for C_0 is defined in Figure 2.9. The C_0 language supports the same operators as R_1 but the arguments of operators are now restricted to just variables and integers, thanks to the `remove-complex-opera*` pass. In the literature this style of intermediate language is called administrative normal form, or ANF for short [**??**]. Instead of `let` expressions, C_0 has assignment statements which can be executed in sequence using the `seq` construct. 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 in tail position depending on the kind of expression.)

A C_0 program consists of an association list mapping labels to tails. This is overkill 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 will be it's tail. The *info* field of the program construt, after the `uncover-locals` pass, will contain a mapping from the symbol `locals` to a list of variables, that is, a list of all the variables used in

$$
\begin{array}{lcl}
\mathit{arg} & ::= & \mathit{int} \mid \mathit{var} \\
\mathit{exp} & ::= & \mathit{arg} \mid (\texttt{read}) \mid (\texttt{-}\ \mathit{arg}) \mid (\texttt{+}\ \mathit{arg}\ \mathit{arg}) \\
\mathit{stmt} & ::= & (\texttt{assign}\ \mathit{var}\ \mathit{exp}) \\
\mathit{tail} & ::= & (\texttt{return}\ \mathit{exp}) \mid (\texttt{seq}\ \mathit{stmt}\ \mathit{tail}) \\
C_0 & ::= & (\texttt{program}\ \mathit{info}\ ((\mathit{label}\ .\ \mathit{tail})^{+}))
\end{array}
$$

Figure 2.9: The C_0 intermediate language.

the program. At the start of the program, these variables are uninitialized (they contain garbage) and each variable becomes initialized on its 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 variables and looser rules regarding instruction arguments. The x86† language, the output of `print-x86`, is the concrete syntax for x86.

2.4 Uniquify Variables

The purpose of this pass is to make sure that each `let` uses a unique variable name. For example, the `uniquify` pass should translate the program on the left into the program on the right.

```
(program ()                        (program ()
  (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`.

```
(program ()                         (program ()
  (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` as a structurally recursive function that mostly 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 association list. The `uniquify` function will need to access this

```
(define (uniquify-exp alist)
  (lambda (e)
    (match e
      [(? symbol?) ___]
      [(? integer?) e]
      [`(let ([,x ,e]) ,body) ___]
      [`(,op ,es ...)
       `(,op ,@(for/list ([e es]) ((uniquify-exp alist) e)))]
      )))

(define (uniquify alist)
  (lambda (e)
    (match e
      [`(program ,info ,e)
       `(program ,info ,((uniquify-exp alist) e))]
      )))
```

Figure 2.10: Skeleton for the `uniquify` pass.

association list when it gets to a variable reference, so we add another parameter to `uniquify` for the association list. It is quite common for a compiler pass to need a map to store extra information about variables. Such maps are often called *symbol tables*.

The skeleton of the `uniquify` function is shown in Figure 2.10. The function is curried so that it is convenient to partially apply it to an association list and then apply it to different expressions, as in the last clause for primitive operations in Figure 2.10. In the last `match` clause for the primitive operators, note the use of the comma-@ operator to splice a list of S-expressions into an enclosing S-expression.

Exercise 2. Complete the `uniquify` pass by filling in the blanks, that is, implement the clauses for variables and for the `let` construct.

Exercise 3. 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` constructs, 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.

2.5 Remove Complex Operators and Operands

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

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

We recommend implementing this pass with two mutually recursive functions, `rco-arg` and `rco-exp`. The idea is to apply `rco-arg` to subexpressions that need to become simple and to apply `rco-exp` to subexpressions can stay complex. Both functions take an expression in R_1 as input. The `rco-exp` function returns an expression. The `rco-arg` function returns two things: a simple expression and association list 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 constructs, the Racket documentation will be of help. Also, the `for/lists` construct is useful for applying a function to each element of a list, in the case where the function returns multiple values.

```
                                  (values 'tmp.1
(rco-arg '(- 10))             ⇒           '((tmp.1 . (- 10))))
```

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

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

and not translate them to the following, which might result from a careless implementation of `rco-exp` and `rco-arg`.

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

Exercise 4. Implement the `remove-complex-opera*` pass and test it 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 all of 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 makes the order of execution explicit in the syntax of the program. For R_1, this amounts to flattening `let` constructs into a sequence of assignment statements. For example, consider the following R_1 program.

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

The output of `remove-complex-opera*` is shown below, on the left. 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 result of `explicate-control` produces code in the C_0 language that makes this explicit.

```
                                   (program ()
(program ()                          ((start .
  (let ([y (let ([x.1 20])            (seq (assign x.1 20)
            (let ([x.2 22])    ⇒      (seq (assign x.2 22)
              (+ x.1 x.2)))])         (seq (assign y (+ x.1 x.2))
   y))                                (return y)))))))
```

We recommend implementing `explicate-control` using two mutually recursive functions: `explicate-control-tail` and `explicate-control-assign`. The `explicate-control-tail` function should be applied to expressions in tail position, whereas `explicate-control-assign` should be applied to expressions that occur on the right-hand-side of a `let`. The function `explicate-control-tail` takes an R_1 expression as input and produces a C_0 *tail* (see the grammar in Figure 2.9). The `explicate-control-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`).

2.7 Uncover Locals

The pass `uncover-locals` simply collects all of the variables in the program and places then in the *info* of the program construct. Here is the output for the example program of the last section.

```
(program ((locals . (x.1 x.2 y)))
  ((start .
   (seq (assign x.1 20)
   (seq (assign x.2 22)
   (seq (assign y (+ x.1 x.2))
   (return y)))))))
```

2.8 Select Instructions

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

The cases for *arg* are straightforward, simply putting variables and integer literals into the s-expression format expected of pseudo-x86, `(var` x`)` and `(int` n`)`, respectively.

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

```
                                   (movq (int 10) (var x))
(assign x (+ 10 32))         ⇒
                                   (addq (int 32) (var x))
```

There are some cases that require special care to avoid generating needlessly complicated code. If one of the arguments 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.

```
(assign x (+ 10 x))          ⇒   (addq (int 10) (var x))
```

The `read` operation does not have a direct counterpart in x86 assembly, so we have instead implemented this functionality in the C language, 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 will need to compile `runtime.c` to `runtime.o` (an "object file", using `gcc` option `-c`) and link it into the final executable. For our purposes of code generation, all you need to do is translate an assignment of `read` to 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.

```
                                 (callq read_int)
(assign lhs (read))          ⇒   (movq (reg rax) (var lhs))
```

There are two cases for the *tail* non-terminal: `return` and `seq`. Regarding (`return` e), 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*), we simply process the statement *s* and tail *t* recursively and append the resulting instructions.

Exercise 5. 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.

2.9 Assign Homes

As discussed in Section 2.3, the `assign-homes` pass places all of the variables on the stack. Consider again the example R_1 program `(+ 52 (- 10))`, which after `select-instructions` looks like the following.

```
(movq (int 10) (var tmp.1))
(negq (var tmp.1))
(movq (var tmp.1) (var tmp.2))
(addq (int 52) (var tmp.2))
(movq (var tmp.2) (reg rax)))
```

The variable `tmp.1` is assigned to stack location `-8(%rbp)`, and `tmp.2` is assign to `-16(%rbp)`, so the `assign-homes` pass translates the above to

```
(movq (int 10) (deref rbp -8))
(negq (deref rbp -8))
(movq (deref rbp -8) (deref rbp -16))
(addq (int 52) (deref rbp -16))
(movq (deref rbp -16) (reg rax)))
```

In the process of assigning stack locations to variables, 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. Some operating systems place restrictions on the frame size. For example, Mac OS X requires the frame size to be a multiple of 16 bytes.

Exercise 6. 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.

2.10 Patch Instructions

The purpose of this pass is to make sure that each instruction adheres to the restrictions regarding which arguments can be memory references. For most instructions, the rule is that at most one argument may be a memory reference.

Consider again the following example.

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

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

```
(movq (int 42) (deref rbp -8))
(movq (deref rbp -8) (deref rbp -16))
(movq (deref rbp -16) (reg 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 to the register `rax` and then from `rax` to the destination, as follows.

```
(movq (int 42) (deref rbp -8))
(movq (deref rbp -8) (reg rax))
(movq (reg rax) (deref rbp -16))
(movq (deref rbp -16) (reg rax))
```

Exercise 7. 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.

2.11 Print x86

The last step of the compiler from R_1 to x86 is to convert the x86 AST (defined in Figure 2.7) to the string representation (defined in Figure 2.3). 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 2.5 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 2.9) 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 8. 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.

3

Register Allocation

In Chapter 2 we simplified the generation of x86 assembly by placing all variables on the stack. We can improve the performance of the generated code considerably if we instead place as many variables as possible into registers. The CPU can access a register in a single cycle, whereas accessing the stack takes many cycles to go to cache or many more to access main memory. 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. It is often the case that we have 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, 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 we 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*, and represent this relation as graph whose vertices are variables and edges indicate when two variables interfere with eachother (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 say that when a variable that is assigned to a stack location,

R_1 program:

```
(program ()
  (let ([v 1])
  (let ([w 46])
  (let ([x (+ v 7)])
  (let ([y (+ 4 x)])
  (let ([z (+ x w)])
       (+ z (- y))))))))
```

After instruction selection:

```
(program
 ((locals . (v w x y z t.1)))
 ((start .
   (block ()
     (movq (int 1) (var v))
     (movq (int 46) (var w))
     (movq (var v) (var x))
     (addq (int 7) (var x))
     (movq (var x) (var y))
     (addq (int 4) (var y))
     (movq (var x) (var z))
     (addq (var w) (var z))
     (movq (var y) (var t.1))
     (negq (var t.1))
     (movq (var z) (reg rax))
     (addq (var t.1) (reg rax))
     (jmp conclusion)))))
```

Figure 3.1: An example program for register allocation.

it has been *spilled*. The process of spilling variables is handled as part of the graph coloring process described in 3.4.

3.1 Registers and Calling Conventions

As we perform register allocation, we will need to be aware of the conventions that govern the way in which registers interact with function calls. 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 saving and restoring some other registers, the *callee-saved registers*, before and after using them. The caller-saved registers are

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

while the callee-saved registers are

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

Another way to think about this caller/callee convention is the following. 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 and restoring the value from the stack.

3.2 Liveness Analysis

A variable is *live* if the variable is used at some later point in the program and there is not an intervening assignment to the variable. To understand the latter condition, 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 (int 5) (var a))
2   (movq (int 30) (var b))
3   (movq (var a) (var c))
4   (movq (int 10) (var b))
5   (addq (var b) (var c))
```

The answer is no because the value 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 variables 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 variables after instruction I_k and $L_{\mathsf{before}}(k)$ for the set of live variables before instruction I_k. The live variables after an instruction are always the same as the live variables before the next instruction.

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

To start things off, there are no live variables after the last instruction, so

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

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

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

where $W(k)$ are the variables written to by instruction I_k and $R(k)$ are the variables read by instruction I_k. Figure 3.2 shows the results of live variables analysis for the running example, with each instruction aligned with its $L_{\mathtt{after}}$ set to make the figure easy to read.

```
(block ()                            | {}
  (movq (int 1) (var v))             | {v}
  (movq (int 46) (var w))            | {v, w}
  (movq (var v) (var x))             | {w, x}
  (addq (int 7) (var x))             | {w, x}
  (movq (var x) (var y))             | {w, x, y}
  (addq (int 4) (var y))             | {w, x, y}
  (movq (var x) (var z))             | {w, y, z}
  (addq (var w) (var z))             | {y, z}
  (movq (var y) (var t.1))           | {z, t.1}
  (negq (var t.1))                   | {z, t.1}
  (movq (var z) (reg rax))           | {t.1}
  (addq (var t.1) (reg rax))         | {}
  (jmp conclusion))                  | {}
```

Figure 3.2: An example block annotated with live-after sets.

Exercise 9. Implement the compiler pass named `uncover-live` that computes the live-after sets. We recommend storing the live-after sets (a list of lists of variables) in the *info* field of the `block` construct. 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 variables that appear in an argument (of an instruction), 2) compute the variables read by an instruction which corresponds to the R function discussed above, and 3) the variables written by an instruction which corresponds to W.

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 variables 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 rather expensive because it takes $O(n^2)$ time to look at every pair in a set of n live variables. Second, there is a special case in which two variables 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. That is, for each instruction, create an edge between the variable being written to and all the *other* live variables. (One should not create self edges.) For a `callq` instruction, think of all caller-saved registers as being written to, so and edge must 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 three rules.

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

2. If instruction I_k is of the form (`callq` *label*), then add an edge (r, v) for every caller-saved register r and every variable $v \in L_{\mathsf{after}}(k)$.

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

Working from the top to bottom of Figure 3.2, we obtain the following interference for the instruction at the specified line number.

Line 2: no interference,
Line 3: w interferes with v,
Line 4: x interferes with w,
Line 5: x interferes with w,
Line 6: y interferes with w,
Line 7: y interferes with w and x,
Line 8: z interferes with w and y,
Line 9: z interferes with y,
Line 10: $t.1$ interferes with z,
Line 11: $t.1$ interferes with z,
Line 12: no interference,
Line 13: no interference.
Line 14: no interference.

The resulting interference graph is shown in Figure 3.3.

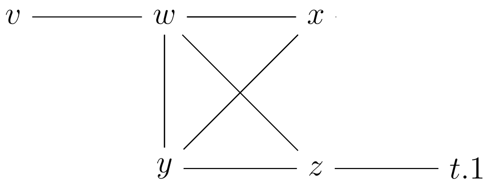


Figure 3.3: The interference graph of the example program.

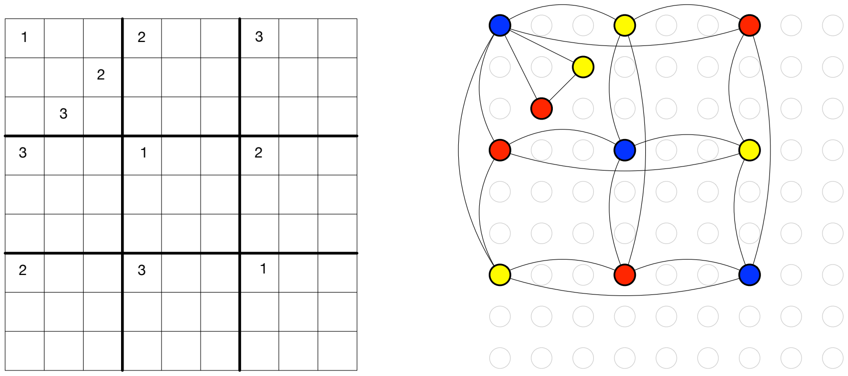
Exercise 10. Implement the compiler pass named `build-interference` according to the algorithm suggested above. We recommend using the Racket `graph` package to create and inspect the interference graph. The output graph of this pass should be stored in the *info* field of the program, under the key `conflicts`.

3.4 Graph Coloring via Sudoku

We now 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 not to map two variables to the same register if the two variables interfere with each other. In terms of 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.



1			2			3		
		2						
	3							
3			1			2		
2			3			1		

Figure 3.4: A Sudoku game board and the corresponding colored 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 3.4 shows 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 (those that are colored) 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 that you 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 color *saturation* due to Brélaz [1979]. The saturation of a vertex, in Sudoku terms, is the set of colors 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

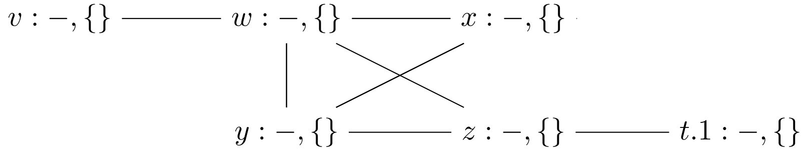
```
Algorithm: DSATUR
Input: a graph $G$
Output: an assignment color[$v$] for each vertex $v \in G$

$W \leftarrow \mathit{vertices}(G)$
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 3.5: The saturation-based greedy graph coloring algorithm.

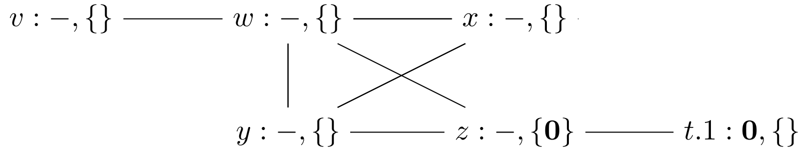
write down that number! But what if there are no squares with only one possibility left? One brute-force approach is to just make a guess. If that guess ultimately leads to a solution, great. If not, backtrack to the guess and make a different guess. One good thing about Pencil Marks is that it reduces 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 making a guess, always choose a square 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 there may not be any possibilities.

In some sense, register allocation is easier than Sudoku because we can always cheat and add more numbers by mapping variables to the stack. We say that a variable is *spilled* when we decide to map it to a stack location. We would like to minimize the time needed to color the graph, and backtracking is expensive. Thus, it makes sense to keep the most-constrained-first heuristic but drop the backtracking in favor of greedy search (guess and just keep going). Figure 3.5 gives the pseudo-code for this simple greedy algorithm for register allocation based on saturation and the most-constrained-first heuristic, which 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, with the first k colors corresponding to the k registers in a given machine and the rest of the integers corresponding to stack locations.

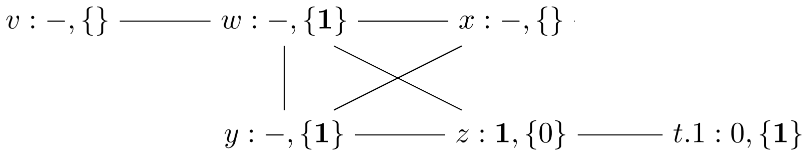
With this algorithm in hand, let us return to the running example and consider how to color the interference graph in Figure 3.3. We shall not use register `rax` for register allocation because we use it to patch instructions, so we remove that vertex from the graph. Initially, all of the vertices are not yet colored and they are unsaturated, so we annotate each of them with a dash for their color and an empty set for the saturation.



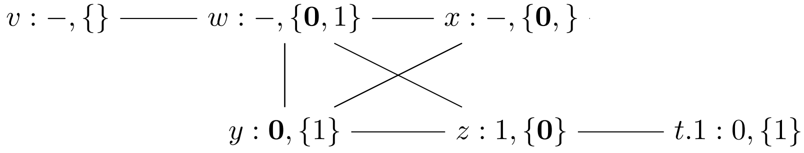
We select a maximally saturated vertex and color it 0. In this case we have a 7-way tie, so we arbitrarily pick $t.1$. The then mark color 0 as no longer available for z because it interferes with $t.1$.



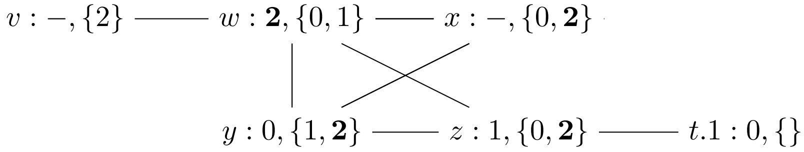
Now we repeat the process, selecting another maximally saturated vertex, which in this case is z. We color z with 1.



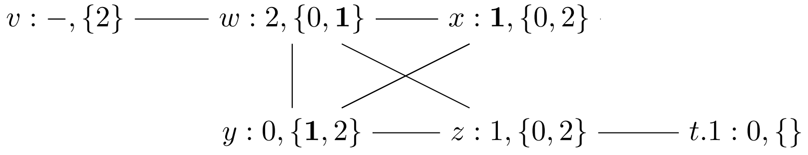
The most saturated vertices are now w and y. We color y with the first available color, which is 0.



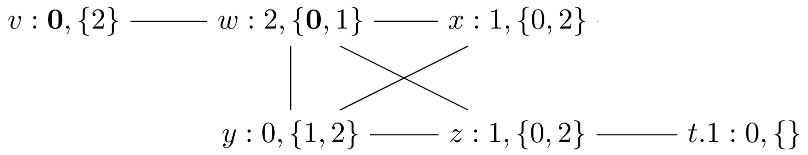
Vertex w is now the most highly saturated, so we color w with 2.


$v : -, \{2\}$
$w : \mathbf{2}, \{0, 1\}$
$x : -, \{0, \mathbf{2}\}$
$y : 0, \{1, \mathbf{2}\}$
$z : 1, \{0, \mathbf{2}\}$
$t.1 : 0, \{\}$

Now x has the highest saturation, so we color it 1.


$v : -, \{2\}$
$w : 2, \{0, \mathbf{1}\}$
$x : \mathbf{1}, \{0, 2\}$
$y : 0, \{\mathbf{1}, 2\}$
$z : 1, \{0, 2\}$
$t.1 : 0, \{\}$

In the last step of the algorithm, we color v with 0.



With the coloring complete, we can finalize the assignment of variables to registers and stack locations. Recall that if we have k registers, 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.

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

Putting this mapping together with the above coloring of the variables, we arrive at the assignment:

$$\{v \mapsto \texttt{\%rcx},\ w \mapsto \texttt{-16(\%rbp)},\ x \mapsto \texttt{-8(\%rbp)},$$
$$y \mapsto \texttt{\%rcx},\ z \mapsto \texttt{-8(\%rbp)}, t.1 \mapsto \texttt{\%rcx}\}$$

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

```
(block ()                        (block ()
  (movq (int 1) (var v))           (movq (int 1) (reg rcx))
  (movq (int 46) (var w))          (movq (int 46) (deref rbp -16))
  (movq (var v) (var x))           (movq (reg rcx) (deref rbp -8))
  (addq (int 7) (var x))           (addq (int 7) (deref rbp -8))
  (movq (var x) (var y))           (movq (deref rbp -8) (reg rcx))
  (addq (int 4) (var y))           (addq (int 4) (reg rcx))
  (movq (var x) (var z))     ⇒     (movq (deref rbp -8) (deref rbp -8))
  (addq (var w) (var z))           (addq (deref rbp -16) (deref rbp -8))
  (movq (var y) (var t.1))         (movq (reg rcx) (reg rcx))
  (negq (var t.1))                 (negq (reg rcx))
  (movq (var z) (reg rax))         (movq (deref rbp -8) (reg rax))
  (addq (var t.1) (reg rax))       (addq (reg rcx) (reg rax))
  (jmp conclusion))                (jmp conclusion))
```

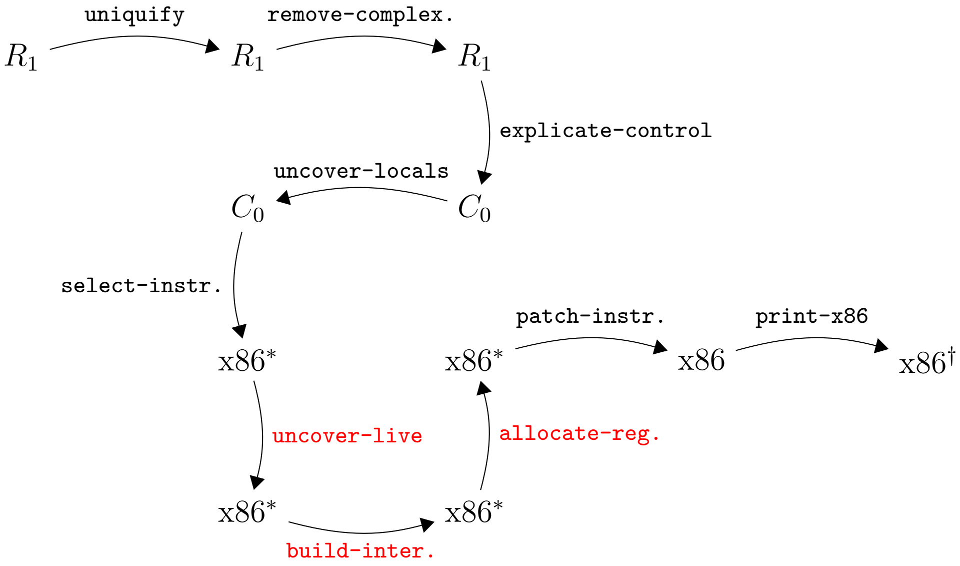


Figure 3.6: Diagram of the passes for R_1 with register allocation.

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

```
(movq (deref rbp -16) (reg rax)
(addq (reg rax) (deref rbp -8))
```

An overview of all of the passes involved in register allocation is shown in Figure 3.6.

Exercise 11. Implement the 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 2.9.

We recommend that you create 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.

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 2.9 to replace the variables with their assigned location.

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 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. In addition, the `main` function needs to restore (in the conclusion) any callee-saved registers that get used during register allocation. 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. 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.

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

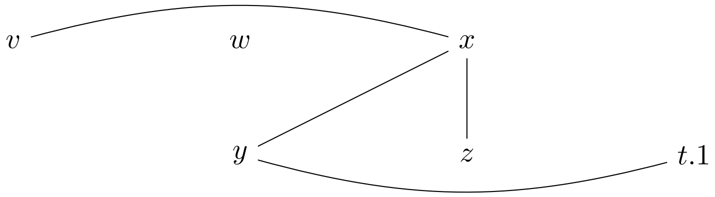
Using the same assignment that was produced by register allocator described in the last section, we get the following program.

```
(block ()                             (block ()
  (movq (int 1) (var v))                (movq (int 1) (reg rbx))
  (movq (int 46) (var w))               (movq (int 46) (reg rdx))
  (movq (var v) (var x))                (movq (reg rbx) (reg rcx))
  (addq (int 7) (var x))                (addq (int 7) (reg rcx))
  (movq (var x) (var y))                (movq (reg rcx) (reg rbx))
  (addq (int 4) (var y))                (addq (int 4) (reg rbx))
  (movq (var x) (var z))         ⇒      (movq (reg rcx) (reg rcx))
  (addq (var w) (var z))                (addq (reg rdx) (reg rcx))
  (movq (var y) (var t.1))              (movq (reg rbx) (reg rbx))
  (negq (var t.1))                      (negq (reg rbx))
  (movq (var z) (reg rax))              (movq (reg rcx) (reg rax))
  (addq (var t.1) (reg rax))            (addq (reg rbx) (reg rax))
  (jmp conclusion))                     (jmp conclusion))
```

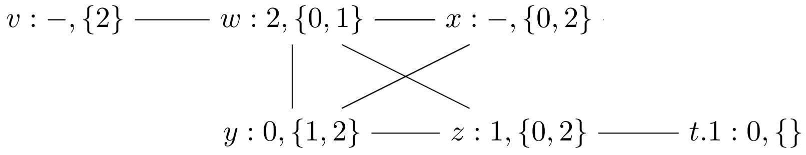
While this allocation is quite good, we could do better. For example, the variables v and x ended up in different registers, but if they had been placed in the same register, then the move from v to x could be removed.

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, but should only be used as a tie breaker when choosing between registers or when choosing between stack locations.

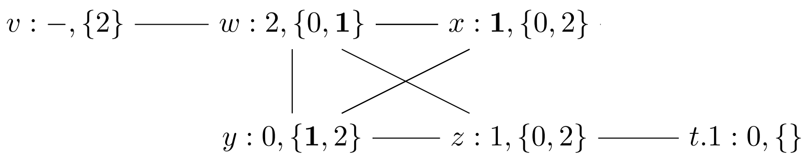
We recommend that you represent the move relationships in a graph, similar to how we represented interference. The following is the *move graph* for our running example.



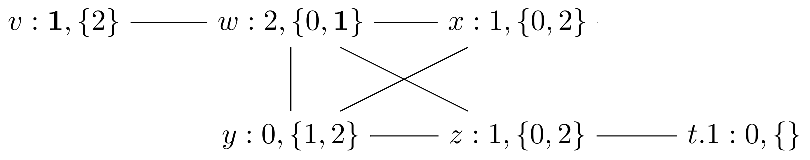
Now we replay the graph coloring, pausing to see the coloring of x and v. So we have the following coloring and the most saturated vertex is x.



Last time we chose to color x with 1, which so happens to be the color of z, and x is move related to z. This was rather lucky, and if the program had been a little different, and say z had been already assigned to 2, then x would still get 1 and our luck would have run out. With move biasing, we use the fact that x and z are move related to influence the choice of color for x, in this case choosing 1 because that's the color of z.


$v : -, \{2\}$
$w : 2, \{0, \mathbf{1}\}$
$x : \mathbf{1}, \{0, 2\}$
$y : 0, \{\mathbf{1}, 2\}$
$z : 1, \{0, 2\}$
$t.1 : 0, \{\}$

Next we consider coloring the variable v, and we just need to avoid choosing 2 because of the interference with w. Last time we choose the color 0, simply because it was the lowest, but this time we know that v is move related to x, so we choose the color 1.



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

```
(block ()                                (block ()
  (movq (int 1) (var v))                   (movq (int 1) (reg rcx))
  (movq (int 46) (var w))                  (movq (int 46) (reg rbx))
  (movq (var v) (var x))                   (movq (reg rcx) (reg rcx))
  (addq (int 7) (var x))                   (addq (int 7) (reg rcx))
  (movq (var x) (var y))                   (movq (reg rcx) (reg rdx))
  (addq (int 4) (var y))                   (addq (int 4) (reg rdx))
  (movq (var x) (var z))              ⇒    (movq (reg rcx) (reg rcx))
  (addq (var w) (var z))                   (addq (reg rbx) (reg rcx))
  (movq (var y) (var t.1))                 (movq (reg rdx) (reg rbx))
  (negq (var t.1))                         (negq (reg rbx))
  (movq (var z) (reg rax))                 (movq (reg rcx) (reg rax))
  (addq (var t.1) (reg rax))               (addq (reg rbx) (reg rax))
  (jmp conclusion))                        (jmp conclusion))
```

The `patch-instructions` then removes the trivial moves from `v` to `x` and from `x` to `z` to obtain the following result.

```
(block ()
  (movq (int 1) (reg rcx))
  (movq (int 46) (reg rbx))
  (addq (int 7) (reg rcx))
  (movq (reg rcx) (reg rdx))
  (addq (int 4) (reg rdx))
  (addq (reg rbx) (reg rcx))
  (movq (reg rdx) (reg rbx))
  (negq (reg rbx))
  (movq (reg rcx) (reg rax))
  (addq (reg rbx) (reg rax))
  (jmp conclusion))
```

Exercise 12. 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.

4

Booleans and Control Flow

The R_0 and R_1 languages only had 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. We also introduce 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 has an impact on several parts of the compiler. Also, because we now have two kinds of values, we need to worry about 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 like `#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 like Racket. The R_2 language is a subset of Typed Racket but by no means

$$
\begin{array}{lcl}
\mathit{cmp} & ::= & \texttt{eq?} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \\
\mathit{exp} & ::= & \mathit{int} \mid (\texttt{read}) \mid (\texttt{-}\ \mathit{exp}) \mid (\texttt{+}\ \mathit{exp}\ \mathit{exp}) \mid (\texttt{-}\ \mathit{exp}\ \mathit{exp}) \\
 & \mid & \mathit{var} \mid (\texttt{let}\ ([\mathit{var}\ \mathit{exp}])\ \mathit{exp}) \\
 & \mid & \texttt{\#t} \mid \texttt{\#f} \mid (\texttt{and}\ \mathit{exp}\ \mathit{exp}) \mid (\texttt{or}\ \mathit{exp}\ \mathit{exp}) \mid (\texttt{not}\ \mathit{exp}) \\
 & \mid & (\mathit{cmp}\ \mathit{exp}\ \mathit{exp}) \mid (\texttt{if}\ \mathit{exp}\ \mathit{exp}\ \mathit{exp}) \\
R_2 & ::= & (\texttt{program}\ \mathit{info}\ \mathit{exp})
\end{array}
$$

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

includes all of Typed Racket. Furthermore, for many of the operations we shall 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 syntax of the R_2 language is defined in Figure 4.1. It 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 subtraction, `and`, `or` and `not`, the `eq?` operations for comparing two integers or two Booleans, and the `<`, `<=`, `>`, and `>=` operations for comparing integers.

Figure 4.2 defines the interpreter for R_2, omitting the parts that are the same as the interpreter for R_1 (Figure 2.2). The literals `#t` and `#f` simply evaluate to themselves. The conditional expression (`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 is somewhat repetitive. In Figure 4.2 we factor out the different parts into the `interp-op` function and the similar parts into the one match clause shown in Figure 4.2. 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`.

As mentioned at the beginning of this chapter, a type checker also rejects programs that apply operators to the wrong type of value. Our type checker for R_2 will signal an error for the following expression because, as we have seen above, the expression `(+ 10 ...)` has type `Integer`, and we require the argument of a **`not`** to have type `Boolean`.

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

The type checker for R_2 is best implemented as a structurally recursive function over the AST. Figure 4.3 shows many of the clauses for the `type-check-exp` function. Given an input expression `e`, the type checker either returns the type (`Integer` or `Boolean`) or it signals an error. Of course, the type of an integer literal is `Integer` and the type of a Boolean literal is `Boolean`. To handle variables, the type checker, like the interpreter, uses an association list. However, in this case the association list maps variables to types instead of values. 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`. When the type checker encounters the use of a variable, it can find its type in the association list.

Exercise 13. Complete the implementation of `type-check-R2` and test it on 10 new example programs in R_2 that you choose based on how thoroughly they test the type checking algorithm. Half of the example programs should have a type error, to make sure that your type checker properly rejects them. The other half of the example programs should not have type errors. Your testing should check that the result of the type checker agrees with

```
(define primitives (set '+ '- 'eq? '< '<= '> '>= 'not 'read))

(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
      ...
      [(? boolean?) e]
      [`(if ,cnd ,thn ,els)
       (define b (recur cnd))
       (match b
         [#t (recur thn)]
         [#f (recur els)])]
      [`(and ,e1 ,e2)
       (define v1 (recur e1))
       (match v1
         [#t (match (recur e2) [#t #t] [#f #f])]
         [#f #f])]
      [`(,op ,args ...)
       #:when (set-member? primitives op)
       (apply (interp-op op) (for/list ([e args]) (recur e)))]
      )))

(define (interp-R2 env)
  (lambda (p)
    (match p
     [`(program ,info ,e)
      ((interp-exp '()) e)])))
```

Figure 4.2: Interpreter for the R_2 language.

```
(define (type-check-exp env)
  (lambda (e)
    (define recur (type-check-exp env))
    (match e
      [(? fixnum?) ’Integer]
      [(? boolean?) ’Boolean]
      [(? symbol? x) (dict-ref env x)]
      [‘(read)     ’Integer]
      [‘(let ([,x ,e]) ,body)
       (define T (recur e))
       (define new-env (cons (cons x T) env))
       (type-check-exp new-env body)]
      ...
      [‘(not ,e)
       (match (recur e)
         [’Boolean ’Boolean]
         [else (error ’type-check-exp "’not’␣expects␣a␣Boolean" e)])]
      ...
      )))

(define (type-check-R2 env)
  (lambda (e)
    (match e
      [‘(program ,info ,body)
       (define ty ((type-check-exp ’()) body))
       ‘(program ,info ,body)]
      )))
```

Figure 4.3: Skeleton of a type checker for the R_2 language.

the value returned by the interpreter, that is, if the type checker returns `Integer`, then the interpreter should return an integer. Likewise, if the type checker returns `Boolean`, then the interpreter should return `#t` or `#f`. 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.

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

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

$$(\texttt{<=}\ e_1\ e_2) \quad\Rightarrow\quad (\texttt{not}\ (\texttt{<}\ e_2\ e_1))$$

By performing these translations near the front-end of the compiler, the later passes of the compiler will 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 14. 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 XOR, Comparisons, and Control Flow in x86

To implement the new logical operations, the comparison operations, and the `if` expression, we need to delve further into the x86 language. Figure 4.4 defines the abstract syntax for a larger subset of x86 that includes instructions for logical operations, comparisons, and 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

$$
\begin{array}{lcl}
\mathit{arg} & ::= & (\texttt{int}\ \mathit{int}) \mid (\texttt{reg}\ \mathit{register}) \mid (\texttt{deref}\ \mathit{register}\ \mathit{int})\\
 & \mid & (\texttt{byte-reg}\ \mathit{register})\\
\mathit{cc} & ::= & \texttt{e} \mid \texttt{l} \mid \texttt{le} \mid \texttt{g} \mid \texttt{ge}\\
\mathit{instr} & ::= & (\texttt{addq}\ \mathit{arg}\ \mathit{arg}) \mid (\texttt{subq}\ \mathit{arg}\ \mathit{arg}) \mid (\texttt{negq}\ \mathit{arg}) \mid (\texttt{movq}\ \mathit{arg}\ \mathit{arg})\\
 & \mid & (\texttt{callq}\ \mathit{label}) \mid (\texttt{pushq}\ \mathit{arg}) \mid (\texttt{popq}\ \mathit{arg}) \mid (\texttt{retq})\\
 & \mid & (\texttt{xorq}\ \mathit{arg}\ \mathit{arg}) \mid (\texttt{cmpq}\ \mathit{arg}\ \mathit{arg}) \mid (\texttt{set}\ \mathit{cc}\ \mathit{arg})\\
 & \mid & (\texttt{movzbq}\ \mathit{arg}\ \mathit{arg}) \mid (\texttt{jmp}\ \mathit{label}) \mid (\texttt{jmp-if}\ \mathit{cc}\ \mathit{label})\\
 & \mid & (\texttt{label}\ \mathit{label})\\
x86_1 & ::= & (\texttt{program}\ \mathit{info}\ (\texttt{type}\ \mathit{type})\ \mathit{instr}^{+})
\end{array}
$$

Figure 4.4: The $x86_1$ language (extends $x86_0$ of Figure 2.7).

`xorq` instruction can be used to encode `not`. The `xorq` instruction takes two arguments, performs a pairwise exclusive-or 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, 0011 XOR 0101 = 0110. Notice that in row of the table for the bit 1, the result is the opposite of the second bit. Thus, the `not` operation can be implemented by `xorq` with 1 as the first argument: 0001 XOR 0000 = 0001 and 0001 XOR 0001 = 0000.

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`, which is 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.

$$
\begin{array}{lcl}
\mathit{arg} & ::= & \mathit{int} \mid \mathit{var} \mid \texttt{\#t} \mid \texttt{\#f} \\
\mathit{cmp} & ::= & \texttt{eq?} \mid \texttt{<} \\
\mathit{exp} & ::= & \mathit{arg} \mid (\texttt{read}) \mid (\texttt{-}\ \mathit{arg}) \mid (\texttt{+}\ \mathit{arg}\ \mathit{arg}) \mid (\texttt{not}\ \mathit{arg}) \mid (\mathit{cmp}\ \mathit{arg}\ \mathit{arg}) \\
\mathit{stmt} & ::= & (\texttt{assign}\ \mathit{var}\ \mathit{exp}) \\
\mathit{tail} & ::= & (\texttt{return}\ \mathit{exp}) \mid (\texttt{seq}\ \mathit{stmt}\ \mathit{tail}) \\
 & \mid & (\texttt{goto}\ \mathit{label}) \mid (\texttt{if}\ (\mathit{cmp}\ \mathit{arg}\ \mathit{arg})\ (\texttt{goto}\ \mathit{label})\ (\texttt{goto}\ \mathit{label})) \\
C_1 & ::= & (\texttt{program}\ \mathit{info}\ ((\mathit{label}\ .\ \mathit{tail})^{+}))
\end{array}
$$

Figure 4.5: The C_1 language, extending C_0 with Booleans and conditionals.

For compiling the `if` expression, the x86 instructions for jumping are relevant. The `jmp` instruction updates the program counter to point to the instruction after the indicated label. The `jmp-if` 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 `jmp-if` instruction falls through to the next instruction. Because the `jmp-if` instruction relies on the EFLAGS register, it is quite common for the `jmp-if` to be immediately preceeded by a `cmpq` instruction, to set the EFLAGS regsiter. Our abstract syntax for `jmp-if` differs from the concrete syntax for x86 to separate the instruction name from the condition code. For example, `(jmp-if le foo)` corresponds to `jle foo`.

4.5 The C_1 Intermediate Language

As with R_1, we shall 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 4.5 shows the new features of C_1; we add logic and comparison operators to the *exp* non-terminal, 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's and conditional goto's in the grammar for *tail*. This means that a sequence of statements may now end with a `goto` or a conditional `goto`, which jumps to one of two labeled pieces of code depending on the outcome of the comparison. In Section 4.6 we discuss how to translate from R_2 to C_1, bridging this gap between `if` expressions and `goto`'s.

4.6 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, things get more interesting.

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

```
(program ()
  (if (if (eq? (read) 1)
          (eq? (read) 0)
          (eq? (read) 2))
      (+ 10 32)
      (+ 700 77)))
```

The naive way to compile `if` and `eq?` would be to handle each of them in isolation, regardless of their context. Each `eq?` 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 `jmp-if`. 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.

```
(if (eq? (read) 1)
    (if (eq? (read) 0)
          (+ 10 32)
          (+ 700 77))
        (if (eq? (read) 2))
          (+ 10 32)
          (+ 700 77))
```

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 `goto`'s 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 ?. 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 represents a control

```
(program ()
  (if (if (eq? (read) 1)
          (eq? (read) 0)
          (eq? (read) 2))
      (+ 10 32)
      (+ 700 77)))
```

⇓

```
(program ()
  (if (if (let ([tmp52 (read)])
            (eq? tmp52 1))
          (let ([tmp53 (read)])
            (eq? tmp53 0))
          (let ([tmp54 (read)])
            (eq? tmp54 2)))
    (+ 10 32)
    (+ 700 77)))
```

⇒

```
(program ()
  ((block62 .
     (seq (assign tmp54 (read))
          (if (eq? tmp54 2)
              (goto block59)
              (goto block60))))
   (block61 .
     (seq (assign tmp53 (read))
          (if (eq? tmp53 0)
               (goto block57)
               (goto block58))))
   (block60 . (goto block56))
   (block59 . (goto block55))
   (block58 . (goto block56))
   (block57 . (goto block55))
   (block56 . (return (+ 700 77)))
   (block55 . (return (+ 10 32)))
   (start .
     (seq (assign tmp52 (read))
          (if (eq? tmp52 1)
               (goto block61)
               (goto block62))))))
```

Figure 4.6: Example translation from R_2 to C_1 via the `explicate-control`.

flow graph as an association list mapping labels to basic blocks. Each block is represented by the *tail* non-terminal.

Figure 4.6 shows the output of the `remove-complex-opera*` pass and then the `explicate-control` pass on the example program. We shall 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 the `(read)` and comparison to `1` from the predicate of the inner `if`. In the output of `explicate-control`, in the `start` block, this becomes a `(read)` followed by a conditional goto to either `block61` or `block62`. Each of these contains the translations of the code `(eq? (read) 0)` and `(eq? (read) 1)`, respectively. Regarding `block61`, we start with the `(read)` and comparison to `0` and then have a conditional goto, either to `block59` or `block60`, which indirectly take us to `block55` and `block56`, the two branches of the outer `if`, i.e., `(+ 10 32)` and `(+ 700 77)`. The story for `block62` is similar.

The nice thing about the output of `explicate-control` is that there are no unnecessary uses of `eq?` and every use of `eq?` is part of a conditional jump. The down-side of this output is that it includes trivial blocks, such as `block57` through `block60`, that only jump to another block. We discuss a solution to this problem in Section 4.11.

Recall that in Section 2.6 we implement the `explicate-control` pass for R_1 using two mutually recursive functions, `explicate-control-tail` and `explicate-control-assign`. The former function translated expressions in tail position whereas the later function translated 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`. So we shall need another function, `explicate-control-pred`, that takes an R_2 expression and two pieces of C_1 code (two *tail*'s) for the then-branch and else-branch. The output of `explicate-control-pred` is a C_1 *tail*. However, these three functions also need to contruct the control-flow graph, which we recommend they do via updates to a global variable. Next we consider the specific additions to the tail and assign functions, and some of cases for the pred function.

The `explicate-control-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-control-tail` on the *thn* branch and B_2 be the result of apply `explicate-control-tail` to the *else* branch. Then the `if` translates to the block B_3 which is the result of applying `explicate-control-pred` to the predicate *cnd* and the blocks B_1 and B_2.

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

Next we consider the case for `if` in the `explicate-control-assign` function. So 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 *thn* and *els* branches shall both need to continue to B_1, so we add B_1 to the control flow graph with a fresh label ℓ_1. Again, the branches of the `if` inherit the current context, so that are in assignment positions. Let B_2 be the result of applying `explicate-control-assign` to the *thn* branch, variable x, and the block `(goto` ℓ_1`)`. Let B_3 be the result of applying `explicate-control-assign` to the *else* branch, variable x, and the block `(goto` ℓ_1`)`. The `if` translates to the block B_4 which is the result of applying `explicate-control-pred` to the predicate *cnd* and the blocks B_2 and B_3.

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

The function `explicate-control-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 branches of the enclosing `if`. One of the base cases of this function is when the expression is a less-than comparision. We translate it to a conditional `goto`. We need labels for the two branches B_1 and B_2, so we add them to the control flow graph and obtain some labels ℓ_1 and ℓ_2. The translation of the less-than comparison is as follows.

$$(\texttt{<}\ e_1\ e_2) \quad\Rightarrow\quad (\texttt{if}\ (\texttt{<}\ e_1\ e_2)\ (\texttt{goto}\ \ell_1)\ (\texttt{goto}\ \ell_2))$$

The case for `if` in `explicate-control-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 input branches B_1 and B_2 to the control flow graph and obtain the labels ℓ_1 and ℓ_2. The branches *thn* and *els* of the current `if` inherit their context from the current one, i.e., predicate context. So we apply `explicate-control-pred` to *thn* with the two blocks `(goto` ℓ_1`)` and `(goto` ℓ_2`)`, to obtain B_3. Similarly for the *els* branch, to obtain B_4. Finally, we apply `explicate-control-pred` to the predicate *cnd* and the blocks B_3 and B_4 to obtain the result B_5.

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

Exercise 15. Implement the pass `explicate-code` by adding the cases for `if` to the functions for tail and assignment contexts, and implement the function for predicate contexts. Create test cases that exercise all of the new cases in the code for this pass.

4.7 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 auxilliary functions, one for each of the non-terminals *arg*, *stmt*, and *tail*.

For *arg*, 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 `(assign` *lhs* `(not` *arg*`))`, if the left-hand side *lhs* is the same as *arg*, then just the `xorq` suffices:

$$(\texttt{assign}\ x\ (\texttt{not}\ x)) \quad\Rightarrow\quad ((\texttt{xorq}\ (\texttt{int}\ 1)\ x'))$$

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

$$(\texttt{assign}\ \mathit{lhs}\ (\texttt{not}\ \mathit{arg})) \quad\Rightarrow\quad ((\texttt{movq}\ \mathit{arg}'\ \mathit{lhs}')\ (\texttt{xorq}\ (\texttt{int}\ 1)\ \mathit{lhs}'))$$

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.

$$(\texttt{assign}\ \mathit{lhs}\ (\texttt{eq?}\ \mathit{arg}_1\ \mathit{arg}_2)) \quad\Rightarrow\quad \begin{array}{l} (\texttt{cmpq}\ \mathit{arg}'_2\ \mathit{arg}'_1) \\ (\texttt{set e (byte-reg al)}) \\ (\texttt{movzbq (byte-reg al)}\ \mathit{lhs}') \end{array}$$

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

$$(\texttt{goto}\ \ell) \quad\Rightarrow\quad ((\texttt{jmp}\ \ell))$$

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").

$$\begin{array}{l} (\texttt{if}\ (\texttt{eq?}\ \mathit{arg}_1\ \mathit{arg}_2) \\ \quad\ (\texttt{goto}\ \ell_1) \\ \quad\ (\texttt{goto}\ \ell_2)) \end{array} \quad\Rightarrow\quad \begin{array}{l} ((\texttt{cmpq}\ \mathit{arg}'_2\ \mathit{arg}'_1) \\ \ (\texttt{jmp-if e}\ \ell_1) \\ \ (\texttt{jmp}\ \ell_2)) \end{array}$$

Exercise 16. 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).

4.8 Register Allocation

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

4.8.1 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` now produces many basic blocks arranged in a control-flow graph. The first question we need to consider is in 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, 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 (it is a DAG, that is, a directed acyclic graph)[1]. What all this amounts to is that we need to process the basic blocks in reverse topological order. We recommend using the `tsort` 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. During compilation we do not know which way the branch will go, so we do not know which of the successor's live-before set to use. The solution comes from the observation that there is no harm in identifying more variables as live than absolutely necessary. Thus, we can take the union of the 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$_1$.

4.8.2 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 not, I recommend that you change your code to be more general. The `movzbq` instruction should be handled like the `movq` instruction.

Exercise 17. 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).

[1] 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 [**?**].

4.9 Patch Instructions

The second argument of the `cmpq` instruction must not be an immediate value (such as a literal 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 `jmp-if`, `jmp`, and `label`.

Exercise 18. 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).

4.10 An Example Translation

Figure 4.7 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 4.8 lists all the passes needed for the compilation of R_2.

4.11 Challenge: Optimize Jumps*

UNDER CONSTRUCTION

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

⇓

```
(program ()
 ((block32 . (return 0))
  (block31 . (return 42))
  (start .
     (seq (assign tmp30 (read))
          (if (eq? tmp30 1)
              (goto block31)
              (goto block32))))))
```

⇓

```
(program ((locals . (tmp30)))
 ((block32 .
    (block ()
      (movq (int 0) (reg rax))
      (jmp conclusion)))
  (block31 .
    (block ()
      (movq (int 42) (reg rax))
      (jmp conclusion)))
  (start .
    (block ()
      (callq read_int)
      (movq (reg rax) (var tmp30))
      (cmpq (int 1) (var tmp30))
      (jmp-if e block31)
      (jmp block32)))))
```

⇒

```
_block31:
        movq    $42, %rax
        jmp _conclusion
_block32:
        movq    $0, %rax
        jmp _conclusion
_start:
        callq   _read_int
        movq    %rax, %rcx
        cmpq    $1, %rcx
        je _block31
        jmp _block32

        .globl _main
_main:
        pushq   %rbp
        movq    %rsp, %rbp
        pushq   %r12
        pushq   %rbx
        pushq   %r13
        pushq   %r14
        subq    $0, %rsp
        jmp _start
_conclusion:
        addq    $0, %rsp
        popq    %r14
        popq    %r13
        popq    %rbx
        popq    %r12
        popq    %rbp
        retq
```

Figure 4.7: Example compilation of an `if` expression to x86.

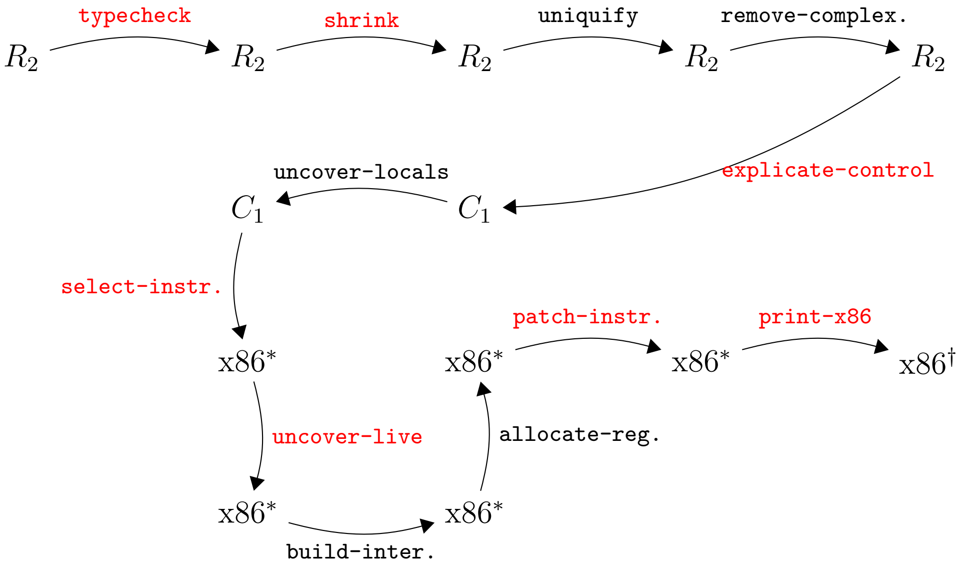


Figure 4.8: Diagram of the passes for R_2, a language with conditionals.

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 implementor'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* 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 5.3 through 5.8 discuss all the necessary changes and additions to the compiler passes, including a new compiler pass named `expose-allocation`.

[1] This may sound contradictory, but Racket's `Void` type corresponds to what is more commonly called the `Unit` type. This type is inhabited by a single value that is usually written `unit` or `()`[?].

```
(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 5.1: Example program that creates tuples and reads from them.

```
type  ::=  Integer | Boolean | (Vector type+) | Void
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)
R3    ::=  (program exp)
```

Figure 5.2: The syntax of R_3, extending R_2 (Figure 4.1) with tuples.

5.1 The R_3 Language

Figure 5.2 defines the syntax for R_3, which includes three new forms for creating a tuple, reading an element of a tuple, and writing to an element of a tuple. The program in Figure 5.1 shows the usage of tuples in Racket. We create a 3-tuple `t` and a 1-tuple. The 1-tuple 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 is taken. The element at index 0 of `t` is 40, to which we add the 2, the element at index 0 of the 1-tuple.

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, i.e., different variables can be *aliases* for the same thing. 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`.

```
(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 the grammar in Figure 5.2 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 `3` even though the variable `t` goes out of scope prior to accessing the vector.

```
(vector-ref
  (let ([t (vector 3 7)])
    t)
  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.

Figure 5.3 shows the definitional interpreter for the R_3 language and Figure 5.4 shows the type checker. The additions to the interpreter are straightforward but the updates to the type checker deserve some explanation. As we shall see in Section 5.2, we need to know which variables are pointers into the heap, that is, which variables are vectors. Also, when allocating a vector, we shall need to know which elements of the vector are pointers. We can obtain this information during type checking and when we uncover local variables. The type checker in Figure 5.4 not only computes the type of an expression, it also wraps every sub-expression e with the form (`has-type` e T), where T is e's type. Subsequently, in the `uncover-locals` pass (Section 5.5) this type information is propagated to all variables (including the temporaries generated by `remove-complex-opera*`).

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].

[2] The R_3 language does not have looping or recursive function, 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-R3 env)
  (lambda (e)
    (match e
      ...
      [else (error 'interp-R3 "unrecognized␣expression")]
      )))
```

Figure 5.3: Interpreter for the R_3 language.

Figure 5.5 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, 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 overapproximate 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*.

```
(define (type-check-exp env)
  (lambda (e)
    (define recur (type-check-exp env))
    (match e
      ...
      ['(void) (values '(has-type (void) Void) 'Void)]
      [`(vector ,es ...)
       (define-values (e* t*) (for/lists (e* t*) ([e es])
                             (recur e)))
       (let ([t `(Vector ,@t*)])
         (debug "vector/type-check-exp␣finished␣vector" t)
         (values `(has-type (vector ,@e*) ,t) t))]
      [`(vector-ref ,e ,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 `(has-type (vector-ref ,e^ (has-type ,i Integer)) ,t)
                  t))]
         [else (error "expected␣a␣vector␣in␣vector-ref,␣not" t)])]
      [`(eq? ,arg1 ,arg2)
       (define-values (e1 t1) (recur arg1))
       (define-values (e2 t2) (recur arg2))
       (match* (t1 t2)
         [(`(Vector ,ts1 ...) `(Vector ,ts2 ...))
          (values `(has-type (eq? ,e1 ,e2) Boolean) 'Boolean)]
         [(other wise) ((super type-check-exp env) e)])]
      ...
      )))
```

Figure 5.4: Type checker for the R_3 language.

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 5.5, there are three pointers in the root 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 5.5) 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 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 sitation in Figure 5.5, with a cycle, cannot be created by a well-typed program in R_3. 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 older siblings, the generational collectors) when its comes to garbage collection, such as mark-and-sweep and reference counting. The strengths of copying collectors are that allocation is fast (just a test 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 disadvantage of two-space copying collectors is that they use a lot of space, though that problem is 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. Of course, 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].

5.2.1 Graph Copying via Cheney's Algorithm

Let us take a closer look at how the copy works. 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

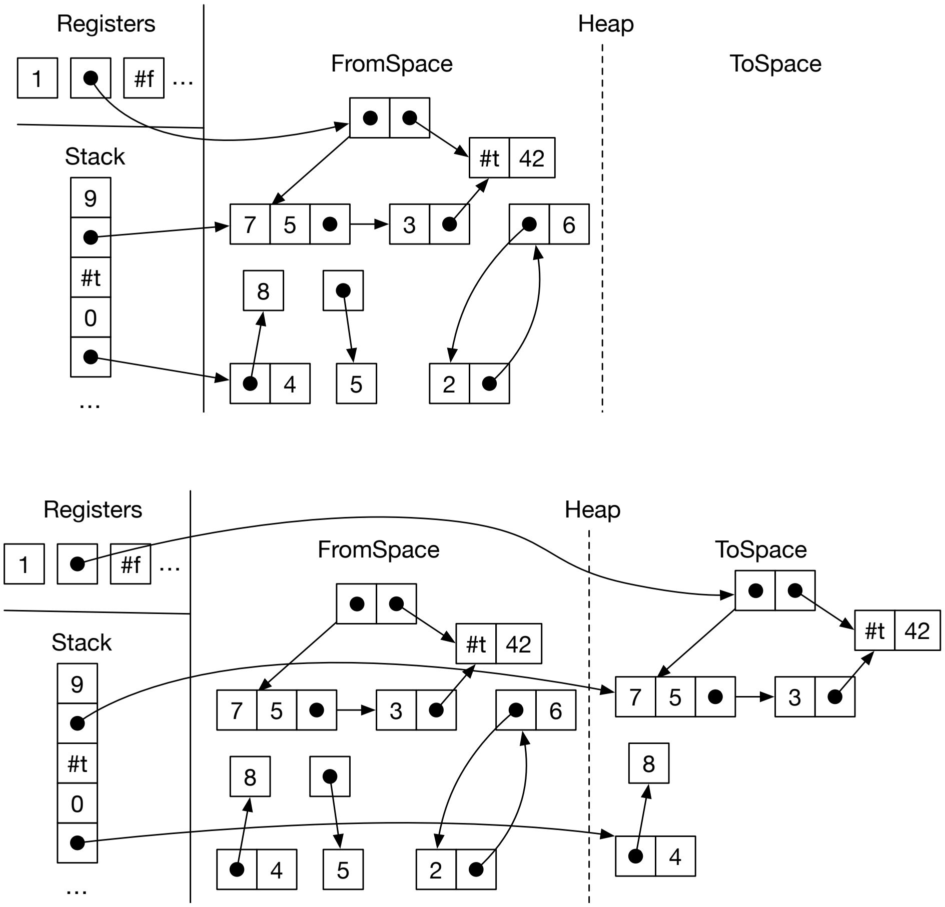


Figure 5.5: A copying collector in action.

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 5.6 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 the marking 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 5.6, 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].

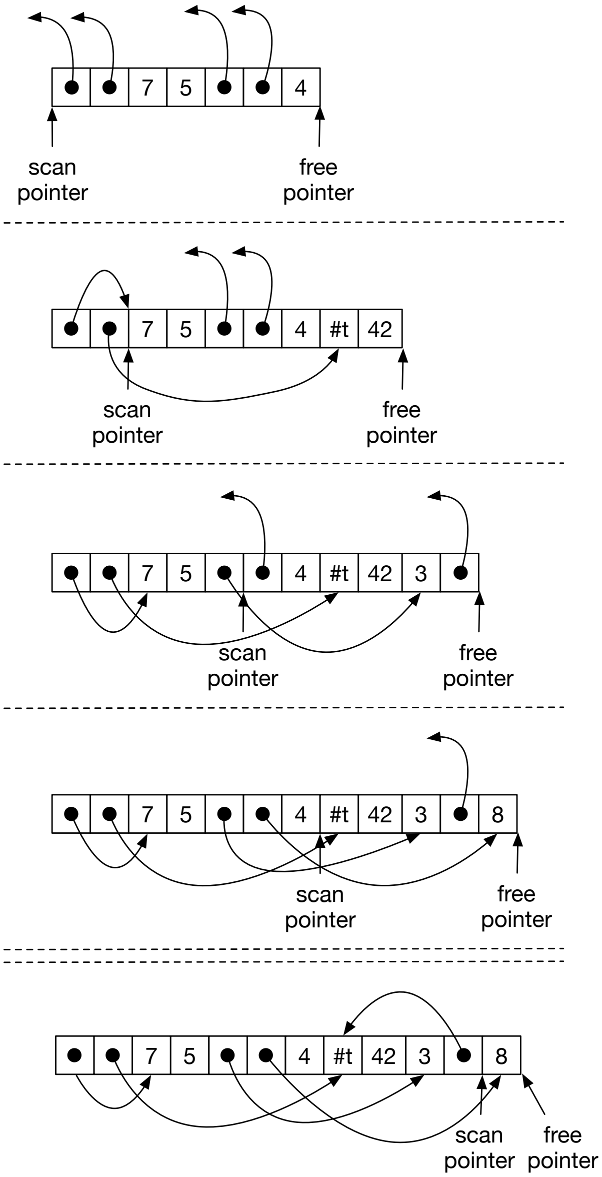


Figure 5.6: Depiction of the Cheney algorithm copying the live tuples.

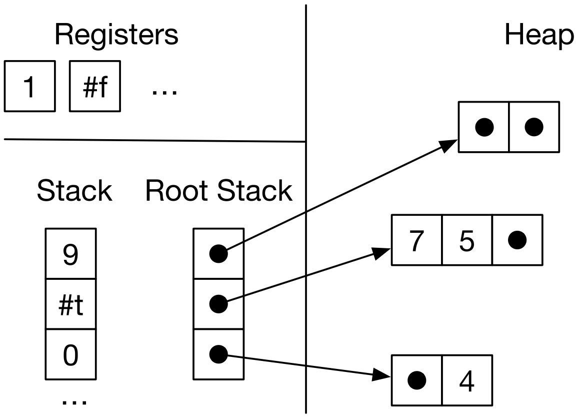


Figure 5.7: Maintaining a root stack to facilitate garbage collection.

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 to a 2-week time budget, we recommend a combination of options 1 and 2, with separate strategies used for the stack and the heap.

Regarding the stack, we recommend using a separate stack for pointers [Siebert, 2001, Henderson, 2002, Baker et al., 2009], which we call a *root stack* (a.k.a. "shadow stack"). 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 5.7 reproduces the example from Figure 5.5 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. We solve this problem by attaching a tag, an extra 64-bits, to each tuple. Figure 5.8 zooms in on the tags for two of the tuples in the example from Figure 5.5. Note that we have drawn the bits in a big-endian way, from right-to-left, with bit location 0 (the least

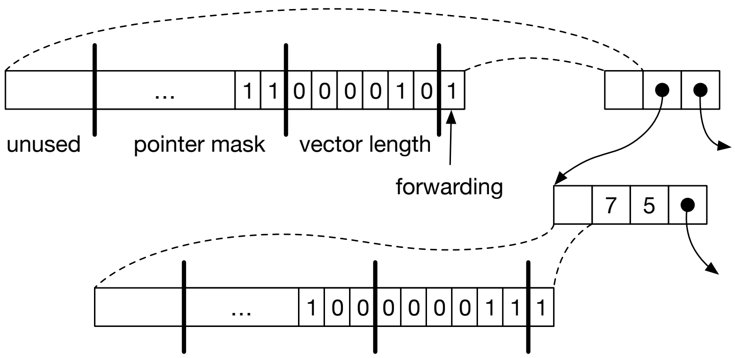


Figure 5.8: Representation for tuples in the heap.

significant bit) on the far right, which corresponds to the directionality 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 bit has value 1, then this tuple has not yet been copied. If the bit has value 0 then the entire tag is in fact a forwarding pointer. (The lower 3 bits of an pointer are always zero anyways because our tuples are 8-byte aligned.)

5.2.3 Implementation of the Garbage Collector

The implementation of the garbage collector needs to do a lot of bit-level data manipulation and we need to link it with our compiler-generated x86 code. Thus, we recommend implementing the garbage collector in C [Kernighan and Ritchie, 1988] and putting the code in the `runtime.c` file. Figure 5.9 shows the interface to the garbage collector. The `initialize` function creates the FromSpace, ToSpace, and root stack. The `initialize` function is meant to be called near the beginning of `main`, before the rest of the program executes. The `initialize` function puts the address of the beginning of the FromSpace into the global variable `free_ptr`. The global `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

```
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 5.9: The compiler's interface to the garbage collector.

`rootstack_begin` global 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.

Exercise 19. *In the file `runtime.c` you will find the implementation of `initialize` and a partial implementation of `collect`. The `collect` function calls another function, `cheney`, to perform the actual copy, and that function is left to the reader to implement. The following is the prototype for `cheney`.*

```
static void cheney(int64_t** rootstack_ptr);
```

The parameter `rootstack_ptr` is a pointer to the top of the rootstack (which is an array of pointers). The `cheney` function also communicates with `collect` through the global variables `fromspace_begin` and `fromspace_end` mentioned in Figure 5.9 as well as the pointers for the ToSpace:

```
static int64_t* tospace_begin;
static int64_t* tospace_end;
```

The job of the `cheney` function is to copy all the live objects (reachable from the root stack) into the ToSpace, update `free_ptr` to point to the next unused spot in the ToSpace, update the root stack so that it points to the objects in the ToSpace, and finally to swap the global pointers for the FromSpace and ToSpace.

The introduction of garbage collection has a non-trivial impact on our compiler passes. We introduce one new compiler pass called `expose-allocation` and make non-trivial changes to `type-check`, `flatten`, `select-instructions`, `allocate-registers`, and `print-x86`. The following program will serve as our running example. It creates two tuples, one nested inside the other. Both tuples have length one. The example then accesses the element in the inner tuple tuple via two vector references.

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

Next we proceed to discuss the new `expose-allocation` pass.

5.3 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 `flatten` because `expose-allocation` introduces new variables, which can be done locally with `let`, but `let` is gone after `flatten`. In the following, we show the transformation for the `vector` form into let-bindings for the intializing expressions, by a conditional `collect`, an `allocate`, and the initialization of the vector. (The *len* is 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!`'s is important, as those expressions may trigger garbage collection and we do not want an allocated but uninitialized tuple to be present during a garbage collection.

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

exp ::= ⋯ | (`collect` *int*) | (`allocate` *int type*) | (`global-value` *name*)

```
(program ()
 (vector-ref
  (vector-ref
   (let ((vecinit48
          (let ((vecinit44 42))
            (let ((collectret46
                   (if (<
                        (+ (global-value free_ptr) 16)
                        (global-value fromspace_end))
                     (void)
                     (collect 16))))
              (let ((alloc43 (allocate 1 (Vector Integer))))
                (let ((initret45 (vector-set! alloc43 0 vecinit44)))
                  alloc43))))))
     (let ((collectret50
            (if (< (+ (global-value free_ptr) 16)
                   (global-value fromspace_end))
              (void)
              (collect 16))))
       (let ((alloc47 (allocate 1 (Vector (Vector Integer)))))
         (let ((initret49 (vector-set! alloc47 0 vecinit48)))
           alloc47))))
   0)
  0))
```

Figure 5.10: Output of the expose-allocation pass, minus all of the has-type forms.

Figure 5.10 shows the output of the expose-allocation pass on our running example.

5.4 Explicate Control and the C_2 language

The output of explicate-control is a program in the intermediate language C_2, whose syntax is defined in Figure 5.11. 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.

```
arg   ::=  int | var | #t | #f
cmp   ::=  eq? | <
exp   ::=  arg | (read) | (- arg) | (+ arg arg) | (not arg) | (cmp arg arg)
       |   (allocate int type) | (vector-ref arg int)
       |   (vector-set! arg int arg) | (global-value name) | (void)
stmt  ::=  (assign var exp) | (return exp) | (collect int)
tail  ::=  (return exp) | (seq stmt tail)
       |   (goto label) | (if (cmp arg arg) (goto label) (goto label))
C_2   ::=  (program info ((label . tail)^+))
```

Figure 5.11: The C_2 language, extending C_1 (Figure 4.5) with vectors.

5.5 Uncover Locals

Recall that the `uncover-locals` function collects all of the local variables so that it can store them in the *info* field of the `program` form. 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 change `uncover-locals` to collect not just the variables, but the variables and their types in the form of an association list. Thanks to the `has-type` forms, the types are readily available. Figure 5.12 lists the output of the `uncover-locals` pass on the running example.

```
(program
 ((locals . ((tmp54 . Integer) (tmp51 . Integer) (tmp53 . Integer)
            (alloc43 . (Vector Integer)) (tmp55 . Integer)
            (initret45 . Void) (alloc47 . (Vector (Vector Integer)))
            (collectret46 . Void) (vecinit48 . (Vector Integer))
            (tmp52 . Integer) (tmp57 . (Vector Integer))
            (vecinit44 . Integer) (tmp56 . Integer) (initret49 . Void)
            (collectret50 . Void))))
 ((block63 . (seq (collect 16) (goto block61)))
  (block62 . (seq (assign collectret46 (void)) (goto block61)))
  (block61 . (seq (assign alloc43 (allocate 1 (Vector Integer)))
              (seq (assign initret45 (vector-set! alloc43 0 vecinit44))
              (seq (assign vecinit48 alloc43)
              (seq (assign tmp54 (global-value free_ptr))
              (seq (assign tmp55 (+ tmp54 16))
              (seq (assign tmp56 (global-value fromspace_end))
              (if (< tmp55 tmp56) (goto block59) (goto block60))))))))
  (block60 . (seq (collect 16) (goto block58)))
  (block59 . (seq (assign collectret50 (void)) (goto block58)))
  (block58 . (seq (assign alloc47 (allocate 1 (Vector (Vector Integer))))
              (seq (assign initret49 (vector-set! alloc47 0 vecinit48))
              (seq (assign tmp57 (vector-ref alloc47 0))
              (return (vector-ref tmp57 0))))))
  (start . (seq (assign vecinit44 42)
          (seq (assign tmp51 (global-value free_ptr))
          (seq (assign tmp52 (+ tmp51 16))
          (seq (assign tmp53 (global-value fromspace_end))
          (if (< tmp52 tmp53) (goto block62) (goto block63)))))))))
```

Figure 5.12: Output of uncover-locals for the running example.

5.6 Select Instructions

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 postpone `global-value` to `print-x86`.

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

```
(assign lhs (vector-ref vec n))
⟹
(movq vec′ (reg r11))
(movq (deref r11 8(n + 1)) lhs)

(assign lhs (vector-set! vec n arg))
⟹
(movq vec′ (reg r11))
(movq arg′ (deref r11 8(n + 1)))
(movq (int 0) lhs)
```

The *vec*′ and *arg*′ are obtained by recursively processing *vec* and *arg*. The move of *vec*′ to register `r11` ensures that offsets are only performed with register operands. This requires removing `r11` from consideration by the register allocating.

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 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 5.8 to see how the tag is organized. We recommend using the Racket operations `bitwise-ior` and `arithmetic-shift` to compute the tag. The type annoation in the `vector` form is used to determine the pointer mask region of the tag.

```
(assign lhs (allocate len (Vector type...)))
⟹
(movq (global-value free_ptr) lhs′)
(addq (int 8(len + 1)) (global-value free_ptr))
(movq lhs′ (reg r11))
(movq (int tag) (deref r11 0))
```

The `collect` form is compiled to a call to the `collect` function in the runtime. The arguments to `collect` are the top of the root stack and the number of bytes that need to be allocated. We shall use a dedicated register,

$$
\begin{array}{lcl}
\mathit{arg} & ::= & (\texttt{int}\ \mathit{int}) \mid (\texttt{reg}\ \mathit{register}) \mid (\texttt{deref}\ \mathit{register}\ \mathit{int}) \\
 & \mid & (\texttt{byte-reg}\ \mathit{register}) \mid (\texttt{global-value}\ \mathit{name}) \\
\mathit{cc} & ::= & \texttt{e} \mid \texttt{l} \mid \texttt{le} \mid \texttt{g} \mid \texttt{ge} \\
\mathit{instr} & ::= & (\texttt{addq}\ \mathit{arg}\ \mathit{arg}) \mid (\texttt{subq}\ \mathit{arg}\ \mathit{arg}) \mid (\texttt{negq}\ \mathit{arg}) \mid (\texttt{movq}\ \mathit{arg}\ \mathit{arg}) \\
 & \mid & (\texttt{callq}\ \mathit{label}) \mid (\texttt{pushq}\ \mathit{arg}) \mid (\texttt{popq}\ \mathit{arg}) \mid (\texttt{retq}) \\
 & \mid & (\texttt{xorq}\ \mathit{arg}\ \mathit{arg}) \mid (\texttt{cmpq}\ \mathit{arg}\ \mathit{arg}) \mid (\texttt{set}\mathit{cc}\ \mathit{arg}) \\
 & \mid & (\texttt{movzbq}\ \mathit{arg}\ \mathit{arg}) \mid (\texttt{jmp}\ \mathit{label}) \mid (\texttt{jmp-if}\mathit{cc}\ \mathit{label}) \\
 & \mid & (\texttt{label}\ \mathit{label}) \\
x86_2 & ::= & (\texttt{program}\ \mathit{info}\ (\texttt{type}\ \mathit{type})\ \mathit{instr}^{+})
\end{array}
$$

Figure 5.13: The $x86_2$ language (extends $x86_1$ of Figure 4.4).

`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)
⟹
(movq (reg r15) (reg rdi))
(movq bytes (reg rsi))
(callq collect)
```

The syntax of the $x86_2$ language is defined in Figure 5.13. It differs from $x86_1$ just in the addition of the form for global variables. Figure 5.14 shows the output of the `select-instructions` pass on the running example.

```
(program
 ((locals . ((tmp54 . Integer) (tmp51 . Integer) (tmp53 . Integer)
            (alloc43 . (Vector Integer)) (tmp55 . Integer)
            (initret45 . Void) (alloc47 . (Vector (Vector Integer)))
            (collectret46 . Void) (vecinit48 . (Vector Integer))
            (tmp52 . Integer) (tmp57 Vector Integer) (vecinit44 . Integer)
            (tmp56 . Integer) (initret49 . Void) (collectret50 . Void))))
 ((block63 . (block ()
              (movq (reg r15) (reg rdi))
              (movq (int 16) (reg rsi))
              (callq collect)
              (jmp block61)))
  (block62 . (block () (movq (int 0) (var collectret46)) (jmp block61)))
  (block61 . (block ()
              (movq (global-value free_ptr) (var alloc43))
              (addq (int 16) (global-value free_ptr))
              (movq (var alloc43) (reg r11))
              (movq (int 3) (deref r11 0))
              (movq (var alloc43) (reg r11))
              (movq (var vecinit44) (deref r11 8))
              (movq (int 0) (var initret45))
              (movq (var alloc43) (var vecinit48))
              (movq (global-value free_ptr) (var tmp54))
              (movq (var tmp54) (var tmp55))
              (addq (int 16) (var tmp55))
              (movq (global-value fromspace_end) (var tmp56))
              (cmpq (var tmp56) (var tmp55))
              (jmp-if l block59)
              (jmp block60)))
  (block60 . (block ()
              (movq (reg r15) (reg rdi))
              (movq (int 16) (reg rsi))
              (callq collect)
              (jmp block58))
  (block59 . (block ()
              (movq (int 0) (var collectret50))
              (jmp block58)))
  (block58 . (block ()
              (movq (global-value free_ptr) (var alloc47))
              (addq (int 16) (global-value free_ptr))
              (movq (var alloc47) (reg r11))
              (movq (int 131) (deref r11 0))
              (movq (var alloc47) (reg r11))
              (movq (var vecinit48) (deref r11 8))
              (movq (int 0) (var initret49))
              (movq (var alloc47) (reg r11))
              (movq (deref r11 8) (var tmp57))
              (movq (var tmp57) (reg r11))
              (movq (deref r11 8) (reg rax))
              (jmp conclusion)))
  (start . (block ()
            (movq (int 42) (var vecinit44))
            (movq (global-value free_ptr) (var tmp51))
            (movq (var tmp51) (var tmp52))
            (addq (int 16) (var tmp52))
            (movq (global-value fromspace_end) (var tmp53))
            (cmpq (var tmp53) (var tmp52))
            (jmp-if l block62)
            (jmp block63))))))
```

Figure 5.14: Output of the `select-instructions` pass.

5.7 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 association list.

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.

5.8 Print x86

Figure 5.15 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 all of 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 accidentaly 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 5.15, 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 5.16 gives an overview of all the passes needed for the compilation of R_3.

```
_block58:
	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
_block59:
	movq	$0, %rcx
	jmp _block58
_block62:
	movq	$0, %rcx
	jmp _block61
_block60:
	movq	%r15, %rdi
	movq	$16, %rsi
	callq	_collect
	jmp _block58
_block63:
	movq	%r15, %rdi
	movq	$16, %rsi
	callq	_collect
	jmp _block61
_start:
	movq	$42, %rbx
	movq	_free_ptr(%rip), %rdx
	addq	$16, %rdx
	movq	_fromspace_end(%rip), %rcx
	cmpq	%rcx, %rdx
	jl _block62
	jmp _block63
```

```
_block61:
	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 _block59
	jmp _block60

	.globl _main
_main:
	pushq	%rbp
	movq	%rsp, %rbp
	pushq	%r12
	pushq	%rbx
	pushq	%r13
	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	%r13
	popq	%rbx
	popq	%r12
	popq	%rbp
	retq
```

Figure 5.15: Output of the `print-x86` pass.

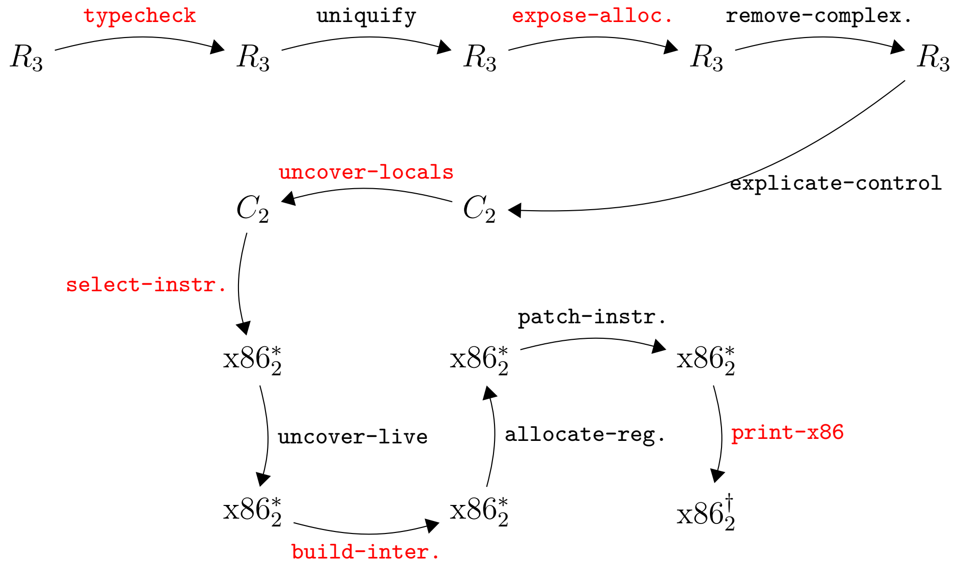


Figure 5.16: Diagram of the passes for R_3, a language with tuples.

6

Functions

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

6.1 The R_4 Language

The syntax for function definitions and function application is shown in Figure 6.1, where we define the R_4 language. Programs in R_4 start 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 syntax for function application does not include an explicit keyword, which is error prone when using `match`. To alleviate this problem, we change the syntax from (*exp exp**) to (`app` *exp exp**) during type checking.

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

`(`$\mathit{type}_1 \cdots \mathit{type}_n$ `->` 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.

$$
\begin{array}{lcl}
\mathit{type} & ::= & \texttt{Integer} \mid \texttt{Boolean} \mid (\texttt{Vector}\ \mathit{type}^{+}) \mid \texttt{Void} \mid (\mathit{type}^{*}\ \texttt{->}\ \mathit{type}) \\
\mathit{cmp} & ::= & \texttt{eq?} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \\
\mathit{exp} & ::= & \mathit{int} \mid (\texttt{read}) \mid (\texttt{-}\ \mathit{exp}) \mid (\texttt{+}\ \mathit{exp}\ \mathit{exp}) \mid (\texttt{-}\ \mathit{exp}\ \mathit{exp}) \\
 & \mid & \mathit{var} \mid (\texttt{let}\ ([\mathit{var}\ \mathit{exp}])\ \mathit{exp}) \\
 & \mid & \texttt{\#t} \mid \texttt{\#f} \mid (\texttt{and}\ \mathit{exp}\ \mathit{exp}) \mid (\texttt{or}\ \mathit{exp}\ \mathit{exp}) \mid (\texttt{not}\ \mathit{exp}) \\
 & \mid & (\mathit{cmp}\ \mathit{exp}\ \mathit{exp}) \mid (\texttt{if}\ \mathit{exp}\ \mathit{exp}\ \mathit{exp}) \\
 & \mid & (\texttt{vector}\ \mathit{exp}^{+}) \mid (\texttt{vector-ref}\ \mathit{exp}\ \mathit{int}) \\
 & \mid & (\texttt{vector-set!}\ \mathit{exp}\ \mathit{int}\ \mathit{exp}) \mid (\texttt{void}) \\
 & \mid & (\mathit{exp}\ \mathit{exp}^{*}) \\
\mathit{def} & ::= & (\texttt{define}\ (\mathit{var}\ [\mathit{var}\texttt{:}\mathit{type}]^{*})\texttt{:}\mathit{type}\ \mathit{exp}) \\
R_4 & ::= & (\texttt{program}\ \mathit{info}\ \mathit{def}^{*}\ \mathit{exp})
\end{array}
$$

Figure 6.1: Syntax of R_4, extending R_3 (Figure 5.2) with functions.

```
(program ()
  (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 6.2: Example of using functions in R_4.

The program in Figure 6.2 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 (a 2-tuple) and returns a new vector containing the results. We also define a function `add1` that does what its name suggests. The program then 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 6.3. The case for the `program` form is responsible for setting up the mutual recursion between the top-level function definitions. We use the classic backpatching approach that uses mutable variables and makes two passes over the function definitions [?]. 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` value's to use the top-level environment.

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 `rip`-relative addressing. For example, the following puts the address of the `add1` label into the `rbx` register.

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

In Section 2.2 we saw the use of the `callq` instruction for jumping to a function whose location is given by a label. Here we instead will be jumping to a function whose location is given by an address, that is, we need to make an *indirect function call*. The x86 syntax is to give the register name prefixed with an asterisk.

```
callq *%rbx
```

6.2.1 Calling Conventions

The `callq` instruction provides partial support for implementing functions, but it does not handle (1) parameter passing, (2) saving and restoring frames on the procedure call stack, 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 shall use the same conventions used by 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`, and `r9`, in that order. 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 shall arrange to never have more than six arguments. The register `rax` is for the return value of the function.

Regarding (2) frames and the procedure call stack, the convention is that the stack grows down, with each function call using a chunk of space called

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

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

(define (interp-R4 p)
  (match p
    [`(program ,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 6.3: Interpreter for the R_4 language.

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

Figure 6.4: Memory layout of caller and callee frames.

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.

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. Thus, the caller should either 1) not put values that are live across a call in caller-saved registers, or 2) save and restore values that are live across calls. We shall recommend option 1). 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. 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. Figure 6.4 shows the 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, $\ldots$, 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 said to be a *tail call*. In such situations, 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 here we've popped 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 parameter passing but instead use 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 preceeding "conclusion" overwrites just about everything else.

```
jmp *%rax
```

6.3 Shrink R_4

The `shrink` pass performs a couple minor modifications to the grammar to ease the later passes. This pass adds an empty *info* field to each function definition:

$$\begin{array}{ll} & \texttt{(define (}f\ \texttt{[}x_1 : \mathit{type}_1\ \texttt{...)}\ \texttt{:}\ \mathit{type}_r\ \mathit{exp}\texttt{)} \\ \Rightarrow & \texttt{(define (}f\ \texttt{[}x_1 : \mathit{type}_1\ \texttt{...)}\ \texttt{:}\ \mathit{type}_r\ \texttt{()}\ \mathit{exp}\texttt{)} \end{array}$$

and introduces an explicit `main` function.

$$\texttt{(program}\ \mathit{info}\ \mathit{ds}\ \texttt{...}\ \mathit{exp}\texttt{)} \quad \Rightarrow \quad \texttt{(program}\ \mathit{info}\ \mathit{ds}'\ \mathit{mainDef}\texttt{)}$$

where *mainDef* is

$$\texttt{(define (main) : Integer ()}\ \mathit{exp}'\texttt{)}$$

6.4 Reveal Functions

Going forward, the syntax of R_4 is inconvenient for purposes of compilation because it conflates the use of function names and local variables. This is a problem 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 `(fun-ref` f`)`. A good name for this pass is `reveal-functions` and the output language, F_1, is defined in Figure 6.5.

Placing this pass after `uniquify` is a good idea, because it 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 will help us compile `fun-ref` into assignment statements.

6.5 Limit Functions

This pass transforms functions so that they have at most six parameters and transforms all function calls so that they pass at most six arguments. A simple strategy for imposing an argument limit of length n is to take all arguments i where $i \geq n$ and pack them into a vector, making that subsequent vector the nth argument.

$$
\begin{array}{lcl}
\mathit{type} & ::= & \texttt{Integer} \mid \texttt{Boolean} \mid (\texttt{Vector}\ \mathit{type}^{+}) \mid \texttt{Void} \mid (\mathit{type}^{*}\ \texttt{->}\ \mathit{type}) \\
\mathit{exp} & ::= & \mathit{int} \mid (\texttt{read}) \mid (\texttt{-}\ \mathit{exp}) \mid (\texttt{+}\ \mathit{exp}\ \mathit{exp}) \\
 & \mid & \mathit{var} \mid (\texttt{let}\ ([\mathit{var}\ \mathit{exp}])\ \mathit{exp}) \\
 & \mid & \texttt{\#t} \mid \texttt{\#f} \mid (\texttt{not}\ \mathit{exp}) \mid (\mathit{cmp}\ \mathit{exp}\ \mathit{exp}) \mid (\texttt{if}\ \mathit{exp}\ \mathit{exp}\ \mathit{exp}) \\
 & \mid & (\texttt{vector}\ \mathit{exp}^{+}) \mid (\texttt{vector-ref}\ \mathit{exp}\ \mathit{int}) \\
 & \mid & (\texttt{vector-set!}\ \mathit{exp}\ \mathit{int}\ \mathit{exp}) \mid (\texttt{void}) \mid (\texttt{app}\ \mathit{exp}\ \mathit{exp}^{*}) \\
 & \mid & (\texttt{fun-ref}\ \mathit{label}) \\
\mathit{def} & ::= & (\texttt{define}\ (\mathit{label}\ [\mathit{var}\texttt{:}\mathit{type}]^{*})\texttt{:}\mathit{type}\ \mathit{exp}) \\
F_1 & ::= & (\texttt{program}\ \mathit{info}\ \mathit{def}^{*})
\end{array}
$$

Figure 6.5: The F_1 language, an extension of R_4 (Figure 6.1).

$$(f\ x_1\ \ldots\ x_n)\ \Rightarrow\ (f\ x_1\ \ldots\ x_5\ (\texttt{vector}\ x_6\ \ldots\ x_n))$$

In the body of the function, all occurrances of the ith argument in which $i > 5$ must be replaced with a `vector-ref`.

6.6 Remove Complex Operators and Operands

The primary decisions to make for this pass is whether to classify `fun-ref` and `app` 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 `app` must be classified as complex expression. Regarding `fun-ref`, as discussed above, the function label needs to be converted to an address using the `leaq` instruction. Thus, even though `fun-ref` 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

Figure 6.6 defines the 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 `fun-ref` and `app`. In assignment and predicate contexts, `app` becomes `call`, whereas in tail position `app` becomes `tailcall`. We recommend defining a new function for processing function definitions. This code is similar to the case

$$
\begin{array}{lcl}
\mathit{arg} & ::= & \mathit{int} \mid \mathit{var} \mid \texttt{\#t} \mid \texttt{\#f} \\
\mathit{cmp} & ::= & \texttt{eq?} \mid \texttt{<} \\
\mathit{exp} & ::= & \mathit{arg} \mid \texttt{(read)} \mid \texttt{(-}\ \mathit{arg}\texttt{)} \mid \texttt{(+}\ \mathit{arg}\ \mathit{arg}\texttt{)} \mid \texttt{(not}\ \mathit{arg}\texttt{)} \mid \texttt{(}\mathit{cmp}\ \mathit{arg}\ \mathit{arg}\texttt{)} \\
 & \mid & \texttt{(allocate}\ \mathit{int}\ \mathit{type}\texttt{)} \mid \texttt{(vector-ref}\ \mathit{arg}\ \mathit{int}\texttt{)} \\
 & \mid & \texttt{(vector-set!}\ \mathit{arg}\ \mathit{int}\ \mathit{arg}\texttt{)} \mid \texttt{(global-value}\ \mathit{name}\texttt{)} \mid \texttt{(void)} \\
 & \mid & \texttt{(fun-ref}\ \mathit{label}\texttt{)} \mid \texttt{(call}\ \mathit{arg}\ \mathit{arg}^*\texttt{)} \\
\mathit{stmt} & ::= & \texttt{(assign}\ \mathit{var}\ \mathit{exp}\texttt{)} \mid \texttt{(return}\ \mathit{exp}\texttt{)} \mid \texttt{(collect}\ \mathit{int}\texttt{)} \\
\mathit{tail} & ::= & \texttt{(return}\ \mathit{exp}\texttt{)} \mid \texttt{(seq}\ \mathit{stmt}\ \mathit{tail}\texttt{)} \\
 & \mid & \texttt{(goto}\ \mathit{label}\texttt{)} \mid \texttt{(if}\ \texttt{(}\mathit{cmp}\ \mathit{arg}\ \mathit{arg}\texttt{)}\ \texttt{(goto}\ \mathit{label}\texttt{)}\ \texttt{(goto}\ \mathit{label}\texttt{))} \\
 & \mid & \texttt{(tailcall}\ \mathit{arg}\ \mathit{arg}^*\texttt{)} \\
\mathit{def} & ::= & \texttt{(define}\ \texttt{(}\mathit{label}\ [\mathit{var}\texttt{:}\mathit{type}]^*\texttt{)}\texttt{:}\mathit{type}\ \texttt{((}\mathit{label}\ \texttt{.}\ \mathit{tail}\texttt{)}^+\texttt{))} \\
C_3 & ::= & \texttt{(program}\ \mathit{info}\ \mathit{def}^*\texttt{)}
\end{array}
$$

Figure 6.6: The C_3 language, extending C_2 (Figure 5.11) with functions.

for `program` in R_3. The top-level `explicate-control` function that handles the `program` form of R_4 can then apply this new function to all the function definitions.

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

The output of select instructions is a program in the x86$_3$ language, whose syntax is defined in Figure 6.7.

An assignment of `fun-ref` becomes a `leaq` instruction as follows:

`(assign` *lhs* `(fun-ref` *f*`))` ⇒ `(leaq (fun-ref` *f*`)` *lhs*`)`

Regarding function definitions, we need to remove their parameters and instead perform parameter passing in terms of the conventions discussed in Section 6.2. That is, the arguments will be in the argument passing

arg	::=	(`int` *int*) | (`reg` *register*) | (`deref` *register int*)
	|	(`byte-reg` *register*) | (`global-value` *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 6.7: The x86$_3$ language (extends x86$_2$ of Figure 5.13).

registers, and inside the function we should generate a `movq` instruction for each parameter, to move the argument value from the appropriate register to a new local variable with the same name as the old parameter.

Next, consider the compilation of function calls, which have the following form upon input to `select-instructions`.

```
(assign lhs (call fun args ...))
```

In the mirror image of handling the parameters of function definitions, the arguments *args* need to be moved to the argument passing registers. Once the instructions for parameter passing have been generated, the function call itself can be performed with an indirect function call, for which I recommend creating the new instruction `indirect-callq`. Of course, the return value from the function is stored in `rax`, so it needs to be moved into the *lhs*.

```
(indirect-callq fun)
(movq (reg 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 `tail-jmp`.

Recall that in Section 2.6 we recommended using the label `start` for the initial block of a program, and in Section 2.8 we recommended labelling 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

Inside `uncover-live`, when computing the W set (written variables) for an `indirect-callq` instruction, we recommend including all the caller-saved registers, which will have the affect of making sure that no caller-saved register actually needs to be saved.

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 5.7 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 `tail-jmp` 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:

```
(fun-ref label) ⇒ label(%rip)
(indirect-callq arg) ⇒ callq *arg
```

Handling `tail-jmp` requires a bit more care. A straightforward translation of `tail-jmp` 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 6.8 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 ommited the `has-type` AST nodes for readability. Can you see any ways to improve the translation?

Exercise 20. Expand your compiler to handle R_4 as outlined in this section. 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 6.9 gives an overview of the passes needed for the compilation of R_4.

```
(program
 (define (add [x : Integer]
              [y : Integer])
   : Integer (+ x y))
 (add 40 2))
```

⇓

```
(program ()
 (define (add86 [x87 : Integer]
         [y88 : Integer]) : Integer ()
   ((add86start . (return (+ x87 y88)))))
 (define (main) : Integer ()
   ((mainstart .
      (seq (assign tmp89 (fun-ref add86))
           (tailcall tmp89 40 2))))))
```

⇒

```
(program ()
 (define (add86)
   ((locals (x87 . Integer) (y88 . Integer))
    (num-params . 2))
   ((add86start .
      (block ()
        (movq (reg rcx) (var x87))
        (movq (reg rdx) (var y88))
        (movq (var x87) (reg rax))
        (addq (var y88) (reg rax))
        (jmp add86conclusion)))))
 (define (main)
   ((locals . ((tmp89 . (Integer Integer -> Integer))))
    (num-params . 0))
   ((mainstart .
      (block ()
        (leaq (fun-ref add86) (var tmp89))
        (movq (int 40) (reg rcx))
        (movq (int 2) (reg rdx))
        (tail-jmp (var tmp89))))))
```

⇓

```
_add90start:
	movq	%rcx, %rsi
	movq	%rdx, %rcx
	movq	%rsi, %rax
	addq	%rcx, %rax
	jmp _add90conclusion
	.globl _add90
	.align 16
_add90:
	pushq	%rbp
	movq	%rsp, %rbp
	pushq	%r12
	pushq	%rbx
	pushq	%r13
	pushq	%r14
	subq	$0, %rsp
	jmp _add90start
_add90conclusion:
	addq	$0, %rsp
	popq	%r14
	popq	%r13
	popq	%rbx
	popq	%r12
	subq $0, %r15
	popq	%rbp
	retq
```

```
_mainstart:
	leaq	_add90(%rip), %rsi
	movq	$40, %rcx
	movq	$2, %rdx
	movq	%rsi, %rax
	addq	$0, %rsp
	popq	%r14
	popq	%r13
	popq	%rbx
	popq	%r12
	subq $0, %r15
	popq	%rbp
	jmp	*%rax

	.globl _main
	.align 16
_main:
	pushq	%rbp
	movq	%rsp, %rbp
	pushq	%r12
	pushq	%rbx
	pushq	%r13
	pushq	%r14
	subq	$0, %rsp
	movq $16384, %rdi
	movq $16, %rsi
	callq _initialize
	movq _rootstack_begin(%rip), %r15
	jmp _mainstart
```

```
_mainconclusion:
	addq	$0, %rsp
	popq	%r14
	popq	%r13
	popq	%rbx
	popq	%r12
	subq $0, %r15
	popq	%rbp
	retq
```

Figure 6.8: Example compilation of a simple function to x86.

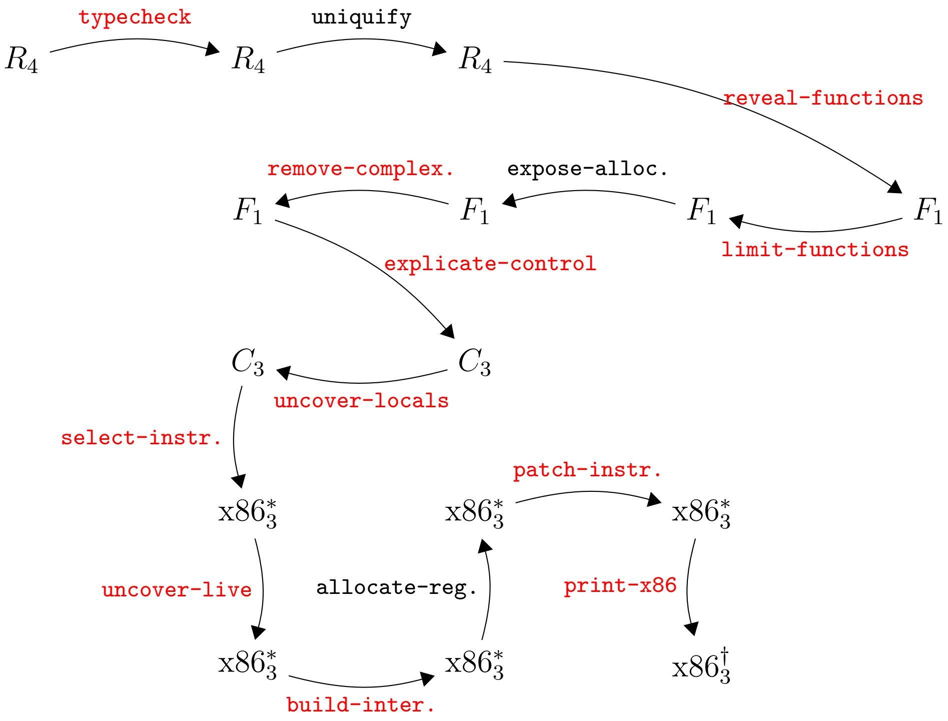


Figure 6.9: Diagram of the passes for R_4, a language with functions.

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 featuring an anonymous function defined 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 `f`. The `lambda` is returned from the function `f`. Below the definition of `f`, we have 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`. Finally, we apply `g` to `11` (producing `20`) and apply `h` to `15` (producing `22`) so the result of this program is `42`.

```
(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.

```
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 7.2: Syntax of R_5, extending R_4 (Figure 6.1) with `lambda`.

7.1 The R_5 Language

The syntax for this language with anonymous functions and lexical scoping, R_5, is defined in Figure 7.2. It adds the `lambda` form to the grammar for R_4, which already has syntax for function application. In this chapter we shall descibe how to compile R_5 back into R_4, compiling lexically-scoped functions into a combination of functions (as in R_4) and tuples (as in R_3).

To compile lexically-scoped functions to top-level function definitions, the compiler will need to provide special treatment to variable occurences such as `x` and `y` in the body of the `lambda` of Figure 7.1, for the functions of R_4 may not refer to variables defined outside the function. To identify such variable occurences, we review the standard notion of free variable.

Definition 21. *A variable is* free with respect to an expression e *if the variable occurs inside* e *but does not have an enclosing binding in* e*.*

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

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

Once we have identified the free variables of a `lambda`, 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. Referring again to Figure 7.1, the binding of `x` to `5` needs to be

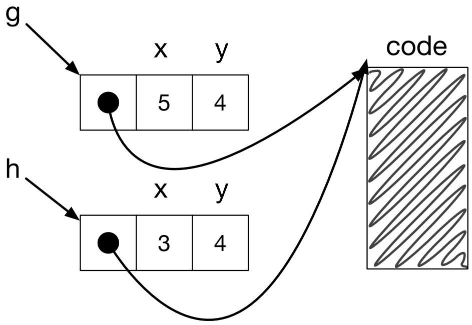


Figure 7.3: Example closure representation for the `lambda`'s in Figure 7.1.

used in the application of `g` to `11`, but the binding of `x` to `3` needs to be used in the application of `h` to `15`. An efficient solution to the problem, due to ?, 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 free variables will fill in the rest of the vector. Figure 7.3 depicts the two closures created by the two calls to `f` in Figure 7.1. Because the two closures came from the same `lambda`, they share the same function pointer but differ in the values for the free variable `x`.

7.2 Interpreting R_5

Figure 7.4 shows the definitional interpreter for R_5. The clause for `lambda` saves the current environment inside the returned `lambda`. Then the clause for `app` 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.

7.3 Type Checking R_5

Figure 7.5 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.

```
(define (interp-exp env)
  (lambda (e)
    (define recur (interp-exp env))
    (match e
      ...
      [‘(lambda: ([,xs : ,Ts] ...) : ,rT ,body)
       ‘(lambda ,xs ,body ,env)]
      [‘(app ,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 7.4: Interpreter for R_5.

```
(define (typecheck-R5 env)
  (lambda (e)
    (match e
      [‘(lambda: ([,xs : ,Ts] ...) : ,rT ,body)
       (define new-env (append (map cons xs Ts) env))
       (define bodyT ((typecheck-R5 new-env) body))
       (cond [(equal? rT bodyT)
              ‘(,@Ts -> ,rT)]
             [else
              (error "mismatch␣in␣return␣type" bodyT rT)])]
      ...
      )))
```

Figure 7.5: Type checking the `lambda`’s in R_5.

7.4 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 `app`. 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 ([fvs_1 (vector-ref clos 1)])
    ...
    (let ([fvs_n (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 retreives 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.

7.5 An Example Translation

Figure 7.6 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 7.6: Example of closure conversion.

Figure 7.7 provides an overview of all the passes needed for the compilation of R_5.

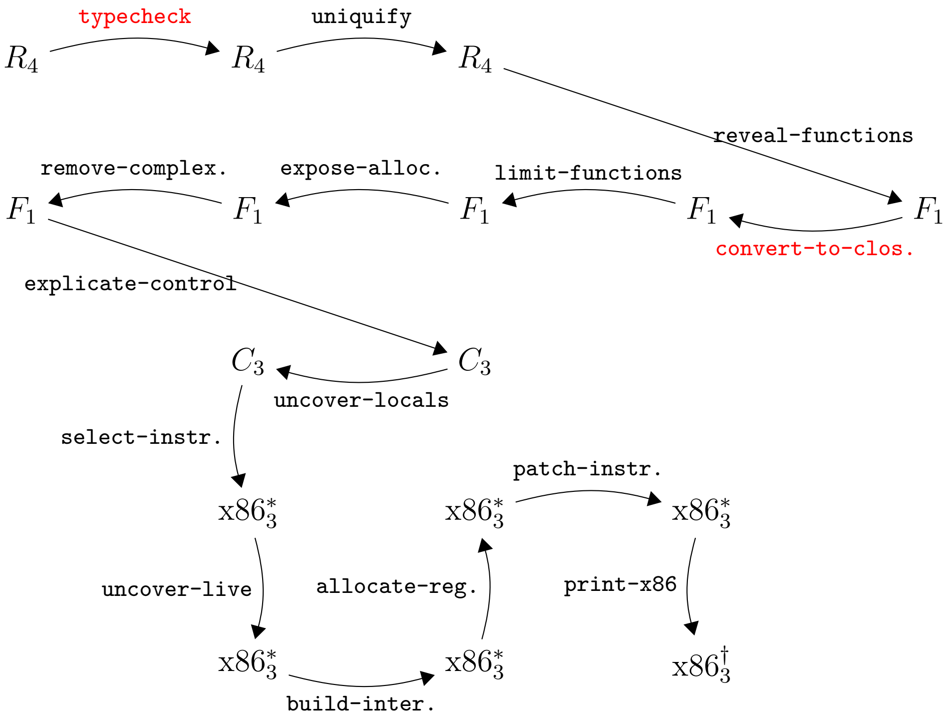


Figure 7.7: Diagram of the passes for R_5, a language with lexically-scoped functions.

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 are talking about here does not have a special name, but it is the usual kind that arrises 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

$$
\begin{array}{rcl}
\mathit{cmp} & ::= & \texttt{eq?} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \\
\mathit{exp} & ::= & \mathit{int} \mid (\texttt{read}) \mid (\texttt{-}\ \mathit{exp}) \mid (\texttt{+}\ \mathit{exp}\ \mathit{exp}) \mid (\texttt{-}\ \mathit{exp}\ \mathit{exp}) \\
 & \mid & \mathit{var} \mid (\texttt{let}\ ([\mathit{var}\ \mathit{exp}])\ \mathit{exp}) \\
 & \mid & \texttt{\#t} \mid \texttt{\#f} \mid (\texttt{and}\ \mathit{exp}\ \mathit{exp}) \mid (\texttt{or}\ \mathit{exp}\ \mathit{exp}) \mid (\texttt{not}\ \mathit{exp}) \\
 & \mid & (\mathit{cmp}\ \mathit{exp}\ \mathit{exp}) \mid (\texttt{if}\ \mathit{exp}\ \mathit{exp}\ \mathit{exp}) \\
 & \mid & (\texttt{vector}\ \mathit{exp}^{+}) \mid (\texttt{vector-ref}\ \mathit{exp}\ \mathit{exp}) \\
 & \mid & (\texttt{vector-set!}\ \mathit{exp}\ \mathit{exp}\ \mathit{exp}) \mid (\texttt{void}) \\
 & \mid & (\mathit{exp}\ \mathit{exp}^{*}) \mid (\texttt{lambda}\ (\mathit{var}^{*})\ \mathit{exp}) \\
 & \mid & (\texttt{boolean?}\ \mathit{exp}) \mid (\texttt{integer?}\ \mathit{exp}) \\
 & \mid & (\texttt{vector?}\ \mathit{exp}) \mid (\texttt{procedure?}\ \mathit{exp}) \mid (\texttt{void?}\ \mathit{exp}) \\
\mathit{def} & ::= & (\texttt{define}\ (\mathit{var}\ \mathit{var}^{*})\ \mathit{exp}) \\
R_7 & ::= & (\texttt{program}\ \mathit{def}^{*}\ \mathit{exp})
\end{array}
$$

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 auxilliary 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.

$$
\begin{array}{lcl}
\mathit{type} & ::= & \texttt{Integer} \mid \texttt{Boolean} \mid (\texttt{Vector}\ \mathit{type}^{+}) \mid (\texttt{Vectorof}\ \mathit{type}) \mid \texttt{Void} \\
 & \mid & (\mathit{type}^{*}\ \texttt{->}\ \mathit{type}) \mid \texttt{Any} \\
\mathit{ftype} & ::= & \texttt{Integer} \mid \texttt{Boolean} \mid \texttt{Void} \mid (\texttt{Vectorof}\ \texttt{Any}) \mid (\texttt{Vector}\ \texttt{Any}^{*}) \\
 & \mid & (\texttt{Any}^{*}\ \texttt{->}\ \texttt{Any}) \\
\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}^{+}) \mid (\texttt{vector-ref}\ \mathit{exp}\ \mathit{int}) \\
 & \mid & (\texttt{vector-set!}\ \mathit{exp}\ \mathit{int}\ \mathit{exp}) \mid (\texttt{void}) \\
 & \mid & (\mathit{exp}\ \mathit{exp}^{*}) \mid (\texttt{lambda:}\ ([\mathit{var}\texttt{:}\mathit{type}]^{*})\texttt{:}\mathit{type}\ \mathit{exp}) \\
 & \mid & (\texttt{inject}\ \mathit{exp}\ \mathit{ftype}) \mid (\texttt{project}\ \mathit{exp}\ \mathit{ftype}) \\
 & \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}\texttt{:}\mathit{type}]^{*})\texttt{:}\mathit{type}\ \mathit{exp}) \\
R_6 & ::= & (\texttt{program}\ \mathit{def}^{*}\ \mathit{exp})
\end{array}
$$

Figure 8.3: Syntax of R_6, extending R_5 (Figure 7.2) 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]))]
    ...))

;; Equavalence 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 ommitted the `has-type` AST nodes to make this output more readable.)

```
                              (let ([tmp e′])
(project e type)      ⇒         (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.

```
                                 (movq e′ lhs')
(assign lhs (inject e T))    ⇒   (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`.

```
                                 (movq e′ lhs')
(assign lhs (inject e T))    ⇒   (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 $\ldots 0111$ (7 in decimal) and apply `bitwise-not` to obtain $\ldots 1000$ 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 avariable of type `Any` is spilled, it must be spilled to the root stack instead of the normal procedure call stack.

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,

#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.

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).

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 several interpreters in the `interp.rkt` file. The `interp-scheme` function takes an AST in one of the Racket-like languages considered in this book ($R_1, R_2, \ldots$) and interprets the program, returning the result value. The `interp-C` function interprets an AST for a program in one of the C-like languages ($C_0, C_1, \ldots$), and the `interp-x86` function interprets an AST for an x86 program.

12.2 Utility Functions

The utility function described in this section can be found in the `utilities.rkt` file.

The `read-program` function takes a file path and parses that file (it must be a Racket program) into an abstract syntax tree (as an S-expression) with a `program` AST at the top.

The `assert` function displays the error message `msg` if the Boolean `bool` is false.

```
(define (assert msg bool) ...)
```

The `lookup` function takes a key and an association list (a list of key-value pairs), and returns the first value that is associated with the given key, if there is one. If not, an error is triggered. The association list may contain both immutable pairs (built with `cons`) and mutable mapirs (built with `mcons`).

The `map2` function ...

12.2.1 Testing

The interp-tests function takes a compiler name (a string), a description of the passes, an interpreter for the source language, a test family name (a string), and a list of test numbers, and runs the compiler passes and the interpreters to check whether the passes correct. The description of the passes is 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. The interpreters from Appendix 12.1 make a good choice. The interp-tests function assumes that the subdirectory tests has a bunch of Scheme programs whose names all start with the family name, followed by an underscore and then the test number, ending in .scm. Also, for each Scheme program there is a file with the same number except that it ends with .in that provides the input for the Scheme program.

```
(define (interp-tests name passes test-family test-nums) ...
```

The compiler-tests function takes a compiler name (a string) a description of the passes (see the comment for interp-tests) a test family name (a string), and a list of test numbers (see the comment for interp-tests), and runs the compiler to generate x86 (a .s file) and then runs gcc to generate machine code. It runs the machine code and checks that the output is 42.

```
(define (compiler-tests name passes test-family test-nums) ...)
```

The compile-file function 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 .scm), applies all of the passes and writes the output to a file whose name is the same as the program file name but with .scm replaced with .s.

```
(define (compile-file passes)
  (lambda (prog-file-name) ...))
```

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	$A \rightarrow B$ (C must be a register)
`cmpq` A, B	compare A and B and set the flag register
`je` L	Jump to label L if the flag register matches the condition code of the instruction, otherwise go to the next instructions. The condition codes are `e` for "equal", `l` for "less", `le` for "less or equal", `g` for "greater", and `ge` for "greater or equal".
`jl` L	
`jle` L	
`jg` L	
`jge` L	
`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	$A \vert B \rightarrow B$ (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	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`).
`setl` A	
`setle` A	
`setg` A	
`setge` A	

Table 12.1: Quick-reference for the x86 instructions used in this book.

Bibliography

Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2nd edition, 1996. ISBN 0262011530.

Hussein Al-Omari and Khair Eddin Sabri. New graph coloring algorithms. *Journal of Mathematics and Statistics*, 2(4), 2006.

AndrewW. 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`.

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. 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.

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`.

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, 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.

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.

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.

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.

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`.

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.

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.