# Essentials of Compilation

## An Incremental Approach

Jeremy G. Siek, Ryan R. Newton
Indiana University

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

January 22, 2018

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

# Contents

# List of Figures

# 1

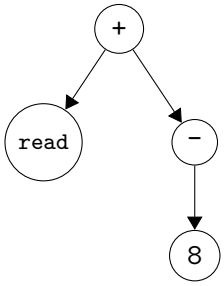
# Preliminaries

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 Scheme, we use S-expressions to represent programs (Section 1.1) and pattern matching to inspect individual nodes in an AST (Section 1.4). We use recursion to construct and deconstruct entire ASTs (Section 1.5).

## 1.1 Abstract Syntax Trees

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.

When deciding how to compile the above program, we need to know that the root node operation is addition and that it has two children: `read` and a negation. The abstract syntax tree data structure directly supports these queries and hence is a good choice. In this book, we will often write down the textual representation of a program even when we really have in mind the AST because the textual representation is more concise. We recommend that, in your mind, you always interpret 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) [**??**]. 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:

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

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.

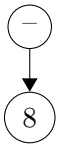
$$exp ::= (\texttt{read}) \tag{1.3}$$

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 for $R_0$ states that there is a `program` node to mark the top of the whole program:

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

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*.

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

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

## 1.3 S-Expressions

Racket, as a descendant of Lisp, has convenient support for creating and manipulating abstract syntax trees with its *symbolic expression* feature, or S-expression for short. We can create an S-expression simply by writing a backquote followed by the textual representation of the AST. (Technically speaking, this is called a *quasiquote* in Racket.) For example, an S-expression to represent the AST (1.1) is created by the following Racket expression:

```
‘(+ (read) (- 8))
```

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.

## 1.4 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)
    (print child2)])
```

```
'+
'(read)
'(- 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 (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))
(leaf? `(- 8))
(leaf? `(+ (read) (- 8)))
```

```
#t
#f
#f
```

## 1.5 Recursion

Programs are inherently recursive in that an $R_0$ *exp* AST is made up of smaller expressions. Thus, the natural way to process in entire program is with a recursive function. As a first example of such a function, we define `R0?` below, which takes an arbitrary S-expression, `sexp`, and determines whether or not `sexp` is in `arith`. Note that each match clause corresponds to one grammar rule for $R_0$ and 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.

```
(define (R0? sexp)
  (define (exp? ex)
    (match ex
      [(? fixnum?) #t]
      [`(read) #t]
      [`(- ,e) (exp? e)]
      [`(+ ,e1 ,e2)
       (and (exp? e1) (exp? e2))]))
  (match sexp
    [`(program ,e) (exp? e)]
    [else #f]))

(R0? `(+ (read) (- 8)))
(R0? `(- (read) (+ 8)))
```

```
#t
#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 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.

## 1.6 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

[1] If you took the *How to Design Programs* course `http://www.ccs.neu.edu/home/matthias/HtDP2e/`, this principle of structuring code according to the data definition is probably quite familiar.

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

Figure 1.2: Interpreter for the $R_0$ language.

report by **?**. The Racket language is defined in its reference manual [**?**]. In this book we use an interpreter to define the meaning of each language that we consider, following Reynold's advice in this regard [**?**]. Here we will warm up by writing an interpreter for the $R_0$ language, which will also serve 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 `exp` helper function, which in turn has one match clause per grammar rule for $R_0$ expressions.

The `exp` function is naturally recursive: clauses for internal AST nodes make recursive calls on each child node. Note that the recursive cases for negation and addition are a place where we could have made use of the `app` feature of Racket's `match` to apply a function and bind the result. The two recursive cases of `interp-R0` would become:

```
[`(- ,(app exp v)) (fx- 0 v)]
[`(+ ,(app exp v1) ,(app exp v2)) (fx+ v1 v2)]))
```

Here we use `(app exp v)` to recursively apply `exp` to the child node and bind the *result value* to variable `v`. The difference between this version and the code in Figure 1.2 is mainly stylistic, although if side effects are involved the order of evaluation can become important. Further, when we write functions with multiple return values, the `app` form can be convenient for binding the resulting values.

Let us consider the result of interpreting some example $R_0$ 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?
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:

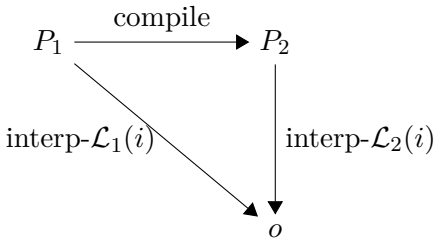
```
42
```

Moving on, the `read` operation prompts the user of the program for an integer. Given an input of `10`, the following program produces `42`.

```
(+ (read) 32)
```

We include the `read` operation in $R_1$ so that a compiler for $R_1$ cannot be implemented simply by running the interpreter at compilation time to obtain the output and then generating the trivial code to return the output. (A clever 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 $o$.




(1.7)

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

```
(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)]
    [`(- ,(app pe-arith r1))
      (pe-neg r1)]
    [`(+ ,(app pe-arith r1) ,(app pe-arith r2))
      (pe-add r1 r2)]))
```

Figure 1.3: A partial evaluator for $R_0$ expressions.

## 1.7 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 the inputs 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))))
```

**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.

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

# 2

# Compiling Integers and Variables

This chapter concerns the challenge of compiling a subset of Racket, which we name $R_1$, to x86-64 assembly code [**?**]. (Henceforth we shall refer to x86-64 simply as x86). The chapter begins with a description of the $R_1$ language (Section 2.1) and then 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 breaking 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 2.8). 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. In addition to variable definitions, the $R_1$ language includes the `program` form to mark the top of the program, which is helpful in some of the compiler passes. The $R_1$ language is rich enough to exhibit several compilation techniques but simple enough so that the reader can implement a compiler for it in a week of part-time work. To give the reader a feeling for the scale of

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

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

this first compiler, the instructor solution for the $R_1$ compiler consists of 6 recursive functions and a few small helper functions that together span 256 lines of code.

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

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

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

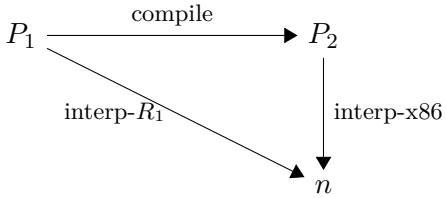
Figure 2.2 shows the interpreter for the $R_1$ language. It extends the interpreter for $R_0$ with two new `match` clauses for variables and for `let`. For `let`, we will need a way to communicate the initializing value of a variable to all the uses of a variable. To accomplish this, we maintain a mapping from variables to values, which is traditionally called an *environment*. For simplicity, here we use an association list to represent the environment. The

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

Figure 2.2: Interpreter for the $R_1$ language.

`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 $n$.




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

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

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

## 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 the program. 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 that comes with the C compiler we use for this course: `gcc`.) 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 reads 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 $d$. The move instruction, `movq` $s$ $d$ reads from $s$ and stores the result in $d$. The

[1]This simple story doesn't fully cover contemporary x86 processors, which combine multiple processing cores per silicon chip, together with hardware memory caches. The result is that, at some instants in real time, different programs may hold conflicting cached values for a given memory address.

```
        .globl main
main:
        movq   $10, %rax
        addq   $32, %rax
        movq   %rax, %rdi
        callq  print_int
        movq   $0, %rax
        retq
```

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

`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`. Finally, the instruction `movq %rax, %rdi` moves the value in `rax` into another register, `rdi`, and `callq print_int` calls the external function `print_int`, which prints the value in `rdi`.

The last two instructions—`movq $0, %rax` and `retq`—finish 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. To ensure that we successfully communicate with the operating system, we explicitly move 0 into `rax`, lest the previous value in `rax` be misinterpreted as an error 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. So the correct output for the above program on Mac would begin with:

```
        .globl _main
_main:
        ...
```

The next example exhibits the use of memory. Figure 2.5 lists an x86 program that is equivalent to (+ 52 (- 10)). To understand how this x86

```
        .globl main
main:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $16, %rsp

        movq    $10, -8(%rbp)
        negq    -8(%rbp)
        movq    $52, %rax
        addq    -8(%rbp), %rax

        movq    %rax, %rdi
        callq   print_int
        addq    $16, %rsp
        movq    $0, %rax
        popq    %rbp
        retq
```

Figure 2.5: An x86 program equivalent to (+ 52 (- 10)).

program works, we need to explain 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. The register `rbp` is the *base pointer* which serves two purposes: 1) it saves the location of the stack pointer for the procedure that called the current one and 2) it is used to access variables associated with the current procedure. 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 $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.

| 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.

The next four instructions 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 last six instructions are the typical *conclusion* of a procedure. The first two print the final result of the program. The latter three are necessary to get the state of the machine back to where it was before the current procedure was called. 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. The `movq $0, %rax` instruction ensures that the returned exit code is 0. Then `popq %rbp` returns the old base pointer to `rbp` and adds 8 to the stack pointer. The `retq` instruction jumps back to the procedure that called this one and subtracts 8 from the stack pointer.

The compiler will need a convenient representation for manipulating x86 programs, so we define an abstract syntax for x86 in Figure 2.7. The *int* field of the `program` AST node is number of bytes of stack space needed for variables in the program. (Some of the intermediate languages will store other information in that location for the purposes of communicating auxiliary data from one step of the compiler to the next. )

## 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. It is these differences that the compiler will need to bridge. What are the differences between $R_1$ and x86 assembly? Here we list some of the most important the differences.

1. x86 arithmetic instructions typically take two arguments and update

| | | |
|---|---|---|
| *register* | ::= | `rsp` \| `rbp` \| `rax` \| `rbx` \| `rcx` \| `rdx` \| `rsi` \| `rdi` \| `r8` \| `r9` \| `r10` \| `r11` \| `r12` \| `r13` \| `r14` \| `r15` |
| *arg* | ::= | (`int` *int*) \| (`reg` *register*) \| (`deref` *register int*) |
| *instr* | ::= | (`addq` *arg arg*) \| (`subq` *arg arg*) \| (`movq` *arg arg*) \| (`retq`) |
| | \| | (`negq` *arg*) \| (`callq` *label*) \| (`pushq` *arg*) \| (`popq` *arg*) |
| $x86_0$ | ::= | (`program` *int* $instr^+$) |

Figure 2.7: Abstract syntax for x86 assembly.

the second argument in place. In contrast, $R_1$ arithmetic operations only read their arguments and produce a new value.

2. An argument to an $R_1$ operator can be any expression, whereas x86 instructions restrict their arguments to integers, registers, and memory locations.

3. An $R_1$ program can have any number of variables whereas x86 has only 16 registers.

4. Variables in $R_1$ can overshadow other variables with the same name. The registers and memory locations of x86 all have unique names.

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. The main question then becomes: in what order do we tackle these differences? This is often one of the most challenging questions that a compiler writer must answer because some orderings may be much more difficult to implement than others. It is difficult to know ahead of time which orders will be better so often some trial-and-error is involved. However, we can try to plan ahead and choose the orderings based on this planning.

For example, to handle difference #2 (nested expressions), we shall introduce new variables and pull apart the nested expressions into a sequence of assignment statements. To deal with difference #3 we will be replacing variables with registers and/or stack locations. Thus, it makes sense to deal with #2 before #3 so that #3 can replace both the original variables and the new ones. Next, consider where #1 should fit in. Because it has to do with the format of x86 instructions, it makes more sense after we have flattened the nested expressions (#2). Finally, when should we deal with #4 (variable overshadowing)? We shall solve this problem by renaming variables to make sure they have unique names. Recall that our plan for #2 involves moving

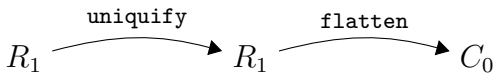
| | | |
|---|---|---|
| *arg* | ::= | *int* \| *var* |
| *exp* | ::= | *arg* \| (`read`) \| (`-` *arg*) \| (`+` *arg arg*) |
| *stmt* | ::= | (`assign` *var exp*) \| (`return` *arg*) |
| $C_0$ | ::= | (`program` (*var*$^*$) *stmt*$^+$) |

Figure 2.8: The $C_0$ intermediate language.

nested expressions, which could be problematic if it changes the shadowing of variables. However, if we deal with #4 first, then it will not be an issue. Thus, we arrive at the following ordering.

$$4 \longrightarrow 2 \longrightarrow 1 \longrightarrow 3$$

We further simplify the translation from $R_1$ to x86 by identifying an intermediate language named $C_0$, roughly half-way between $R_1$ and x86, to provide a rest stop along the way. We name the language $C_0$ because it is vaguely similar to the *C* language [**?**]. The differences #4 and #1, regarding variables and nested expressions, will be handled by two steps, `uniquify` and `flatten`, which bring us to $C_0$.

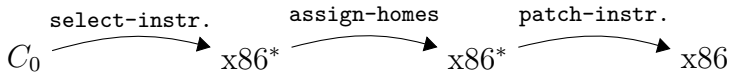



Each of these steps in the compiler is implemented by a function, typically a structurally recursive function that translates an input AST into an output AST. We refer to such a function as a *pass* because it makes a pass over, i.e. it traverses the entire AST.

The syntax for $C_0$ is defined in Figure 2.8. The $C_0$ language supports the same operators as $R_1$ but the arguments of operators are now restricted to just variables and integers. The `let` construct of $R_1$ is replaced by an assignment statement and there is a `return` construct to specify the return value of the program. A program consists of a sequence of statements that include at least one `return` statement. Each program is also annotated with a list of variables (viz. `(var*)`). At the start of the program, these variables are uninitialized (they contain garbage) and each variable becomes initialized on its first assignment. All of the variables used in the program must be present in this list exactly once.

To get from $C_0$ to x86 assembly it remains for us to handle difference #1 (the format of instructions) and difference #3 (variables versus registers). These two differences are intertwined, creating a bit of a Gordian Knot.

To handle difference #3, we need to map some variables to registers (there are only 16 registers) and the remaining variables to locations on the stack (which is unbounded). To make good decisions regarding this mapping, we need the program to be close to its final form (in x86 assembly) so we know exactly when which variables are used. After all, variables that are used in disjoint parts of the program can be assigned to the same register. However, our choice of x86 instructions depends on whether the variables are mapped to registers or stack locations, so we have a circular dependency. We cut this knot by doing an optimistic selection of instructions in the `select-instructions` pass, followed by the `assign-homes` pass to map variables to registers or stack locations, and conclude by finalizing the instruction selection in the `patch-instructions` pass.




The `select-instructions` pass is optimistic in the sense that it treats variables as if they were all mapped to registers. The `select-instructions` pass generates a program that consists of x86 instructions but that still uses variables, so it is an intermediate language that is technically different than x86, which explains the asterisks in the diagram above.

In this Chapter we shall take the easy road to implementing `assign-homes` and simply map all variables to stack locations. The topic of Chapter 3 is implementing a smarter approach in which we make a best-effort to map variables to registers, resorting to the stack only when necessary.

Once variables have been assigned to their homes, we can finalize the instruction selection by dealing with an idiosyncrasy of x86 assembly. Many x86 instructions have two arguments but only one of the arguments may be a memory reference (and the stack is a part of memory). Because some variables may get mapped to stack locations, some of our generated instructions may violate this restriction. The purpose of the `patch-instructions` pass is to fix this problem by replacing every violating instruction with a short sequence of instructions that use the `rax` register. Once we have implemented a good register allocator (Chapter 3), the need to patch instructions will be relatively rare.

## 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 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 2.9. 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 2.9. 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.

```
(define (uniquify alist)
  (lambda (e)
    (match e
      [(? symbol?) ___]
      [(? integer?) e]
      [`(let ([,x ,e]) ,body) ___]
      [`(program ,e)
       `(program ,((uniquify alist) e))]
      [`(,op ,es ...)
       `(,op ,@(map (uniquify alist) es))]
      )))
```

Figure 2.9: Skeleton for the `uniquify` pass.

## 2.5 Flatten Expressions

The `flatten` pass will transform $R_1$ programs into $C_0$ programs. In particular, the purpose of the `flatten` pass is to get rid of nested expressions, such as the `(- 10)` in the program below. This can be accomplished by introducing a new variable, assigning the nested expression to the new variable, and then using the new variable in place of the nested expressions, as shown in the output of `flatten` on the right.

```
(program                          ⇒   (program (tmp.1 tmp.2)
  (+ 52 (- 10)))                        (assign tmp.1 (- 10))
                                        (assign tmp.2 (+ 52 tmp.1))
                                        (return tmp.2))
```

The clause of `flatten` for `let` is straightforward to implement as it just requires the generation of an assignment statement for the `let`-bound variable. The following shows the result of `flatten` for a `let`.

```
(program                                (program (tmp.1 x tmp.2)
  (let ([x (+ (- 10) 11)])        ⇒       (assign tmp.1 (- 10))
    (+ x 41)))                            (assign x (+ tmp.1 11))
                                          (assign tmp.2 (+ x 41))
                                          (return tmp.2))
```

We recommend implementing `flatten` as a structurally recursive function that returns three things, 1) the newly flattened expression, 2) a list of assignment statements, one for each of the new variables introduced during the flattening the expression, and 3) a list of all the variables including both let-bound variables and the generated temporary variables. The newly flattened expression should be an *arg* in the $C_0$ syntax (Figure 2.8), that is, it

should be an integer or a variable. You can return multiple things from a function using the `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 `map3` function (Appendix 12.2) is useful for applying a function to each element of a list, in the case where the function returns three values. The result of `map3` is three lists.

The clause of `flatten` for the `program` node needs to recursively flatten the body of the program and the newly flattened expression should be placed in a `return` statement. Remember that the variable list in the `program` node should contain no duplicates.

Take special care for programs such as the following that initialize variables with integers or other variables. It should be translated to the program on the right

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

$\Rightarrow$

```
(program (a b)
  (assign a 42)
  (assign b a)
  (return b))
```

and not to the following, which could result from a naive implementation of `flatten`.

```
(program (tmp.1 a tmp.2 b)
  (assign tmp.1 42)
  (assign a tmp.1)
  (assign tmp.2 a)
  (assign b tmp.2)
  (return b))
```

**Exercise 4.** Implement the `flatten` 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 `flatten` pass. Use the `interp-tests` function (Appendix 12.2) from `utilities.rkt` to test your passes on the example programs.

## 2.6 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. Also, the `program` form should still list the variables

(similar to $C_0$):

$$(\texttt{program}\ (\mathit{var}^*)\ \mathit{instr}^+)$$

The `select-instructions` pass deals with the differing format of 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 in-place update. So we must first move `10` to `x`.

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

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

Regarding the (`return` *arg*) statement of $C_0$, we recommend treating it as an assignment to the `rax` register and let the procedure conclusion handle the transfer of control back to the calling procedure.

**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.

## 2.7 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 first field of the `program` node which will be needed later to generate the procedure conclusion.

$$(\texttt{program}\ \mathit{int}\ \mathit{instr}^{+})$$

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. I 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.

## 2.8 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))
```

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.

## 2.9 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, for which it is suggest that you compute in the `assign-homes` pass (Section 2.7) and store in the *info* field of the `program` node.

Your compiled code should print the result of the program's execution by using the `print_int` function provided in `runtime.c`. If your compiler has been implemented correctly so far, this final result should be stored in the `rax` register. We'll talk more about how to perform function calls with arguments in general later on, but for now, make sure that your x86 printer includes the following code as part of the conclusion:

```
movq %rax, %rdi
callq print_int
```

These lines move the value in `rax` into the `rdi` register, which stores the first argument to be passed into `print_int`.

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`. In addition to placing underscores on `main`, you need to put them in front of `callq` labels (so `callq print_int` becomes `callq _print_int`).

**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.

Figure 2.10 provides an overview of all the compiler passes described in this Chapter. The x86$^*$ language extends x86 with variables and looser rules regarding instruction arguments. The x86$^\dagger$ language is the concrete syntax (string) for x86.

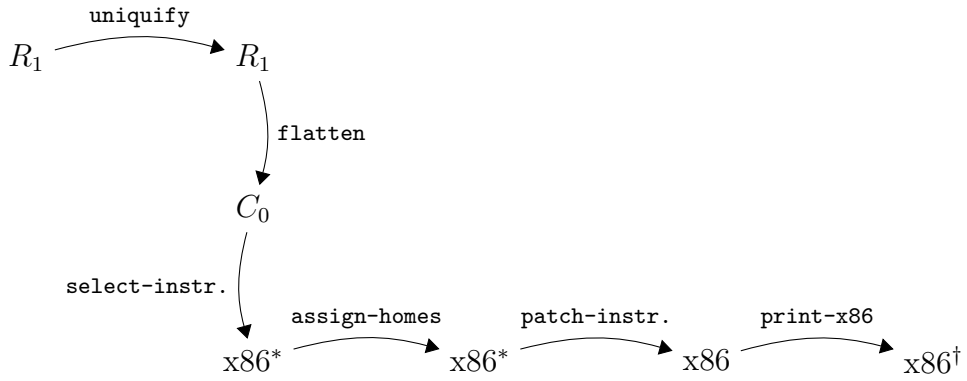



Figure 2.10: Overview of the passes for compiling $R_1$.

# 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 try to 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 3.1 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 3.2). We then model register allocation as a graph coloring problem, which we discuss in Section 3.3.

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,

Source 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 (v w x y z t.1 t.2)
  (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) (var t.2))
  (addq (var t.1) (var t.2))
  (movq (var t.2) (reg rax)))
```

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 3.3.

## 3.1 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

| | | |
|---|---|---|
| 1 | `(program (v w x y z t.1 t.2)` | |
| 2 | `  (movq (int 1) (var v))` | $\{v\}$ |
| 3 | `  (movq (int 46) (var w))` | $\{v, w\}$ |
| 4 | `  (movq (var v) (var x))` | $\{w, x\}$ |
| 5 | `  (addq (int 7) (var x))` | $\{w, x\}$ |
| 6 | `  (movq (var x) (var y))` | $\{w, x, y\}$ |
| 7 | `  (addq (int 4) (var y))` | $\{w, x, y\}$ |
| 8 | `  (movq (var x) (var z))` | $\{w, y, z\}$ |
| 9 | `  (addq (var w) (var z))` | $\{y, z\}$ |
| 10 | `  (movq (var y) (var t.1))` | $\{t.1, z\}$ |
| 11 | `  (negq (var t.1))` | $\{t.1, z\}$ |
| 12 | `  (movq (var z) (var t.2))` | $\{t.1, t.2\}$ |
| 13 | `  (addq (var t.1) (var t.2))` | $\{t.2\}$ |
| 14 | `  (movq (var t.2) (reg rax)))` | $\{\}$ |

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

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.

**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 `program` node alongside the list of variables as follows.

`(program (`$var^*$ $live{-}afters$`)` $instr^+$`)`

I recommend organizing your code to use a helper function that takes a list of statements and an initial live-after set (typically empty) and returns the

list of statements and the list of live-after sets. For this chapter, returning the list of statements is unnecessary, as they will be unchanged, but in Chapter 4 we introduce `if` statements and will need to annotate them with the live-after sets of the two branches.

I 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 $W$.

## 3.2 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 given by the following.

- 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 $v = s$.
- If instruction $I_k$ is not a move but some other arithmetic instruction such as (`addq` $s$ $d$), then add the edge $(d, v)$ for every $v \in L_{\mathsf{after}}(k)$ unless $v = d$.
- If instruction $I_k$ is of the form (`callq` *label*), then add an edge $(r, v)$ for every caller-save register $r$ and every variable $v \in L_{\mathsf{after}}(k)$.

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

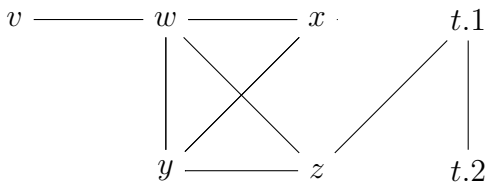



Figure 3.3: The interference graph of the example program.

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: $t.2$ interferes with $t.1$,
Line 13: no interference.
Line 14: no interference.

The resulting interference graph is shown in Figure 3.3.

Our next concern is to choose a data structure for representing the interference graph. There are many standard choices for how to represent a graph: *adjacency matrix*, *adjacency list*, and *edge set* [**?**]. The right way to choose a data structure is to study the algorithm that uses the data structure, determine what operations need to be performed, and then choose the data structure that provide the most efficient implementations of those operations. Often times the choice of data structure can have an effect on the time complexity of the algorithm, as it does here. If you skim the next section, you will see that the register allocation algorithm needs to ask the graph for all of its vertices and, given a vertex, it needs to known all of the adjacent vertices. Thus, the correct choice of graph representation is that of an adjacency list. There are helper functions in `utilities.rkt` for representing graphs using the adjacency list representation: `make-graph`, `add-edge`, and `adjacent` (Appendix 12.2). In particular, those functions use a hash table to map each vertex to the set of adjacent vertices, and the sets are represented using Racket's `set`, which is also a hash table.

**Exercise 10.** Implement the compiler pass named `build-interference` according to the algorithm suggested above. The output of this pass should replace the live-after sets with the interference *graph* as follows.

`(program (`*var*$^*$ *graph*`)` *instr*$^+$`)`

## 3.3 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 we cannot map adjacent nodes to the same register. If we think of registers as colors, the register allocation problem becomes the widely-studied graph coloring problem [**??**].

The reader may be more familiar with the graph coloring problem then 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 node in the graph for each Sudoku square.
- There is an edge between two nodes if the corresponding squares are in the same row, in the same column, or if the squares are in the same $3 \times 3$ 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 nodes in the graph.

If you can color the remaining nodes 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 graph coloring, one can use Sudoku strategies to come up with an algorithm for allocating registers. For example, one of

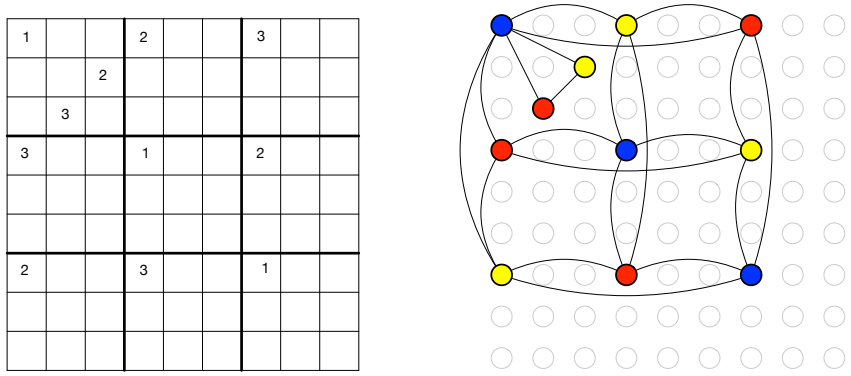



Figure 3.4: A Sudoku game board and the corresponding colored graph.

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. This heuristic also reduces the degree of branching in the search tree.

The Pencil Marks technique corresponds to the notion of color *saturation* due to **?**. The saturation of a node, in Sudoku terms, is the set of colors that are no longer available. In graph terminology, we have the following definition:

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

where adjacent($u$) is the set of nodes adjacent to $u$.

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 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. Of course, backtracking can be horribly time consuming. One standard 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 node with the highest

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

$W \leftarrow \mathit{vertices}(G)$
**while** $W \neq \emptyset$ **do**
    pick a node $u$ 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[$u$] $\leftarrow c$
    $W \leftarrow W - \{u\}$

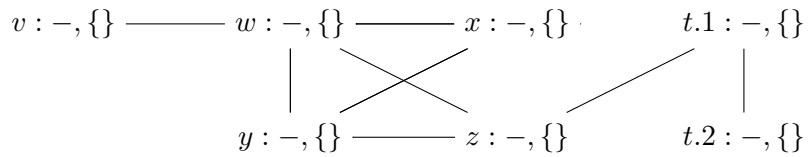
Figure 3.5: The saturation-based greedy graph coloring algorithm.

saturation). The idea is that choosing highly constrained squares earlier rather than later is better because later there may not be any possibilities.

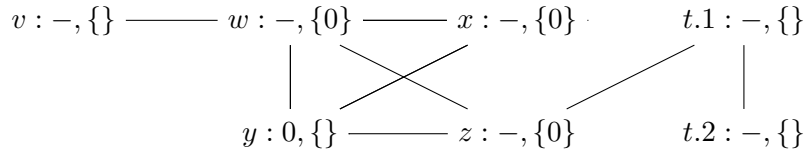
In some sense, register allocation is easier than Sudoku because we can always cheat and add more numbers by mapping variables to the stack. We say that a variable is *spilled* when we decide to map it to a stack location. We would like to minimize the time needed to color the graph, and backtracking 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 **?** (also known as saturation degree ordering [**??**]). 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 nodes are not yet colored and they are unsaturated, so we annotate each of them with a

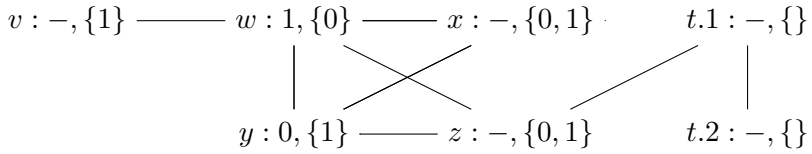
dash for their color and an empty set for the saturation.



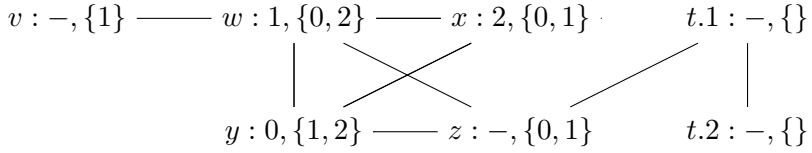

We select a maximally saturated node and color it 0. In this case we have a 7-way tie, so we arbitrarily pick $y$. The then mark color 0 as no longer available for $w$, $x$, and $z$ because they interfere with $y$.



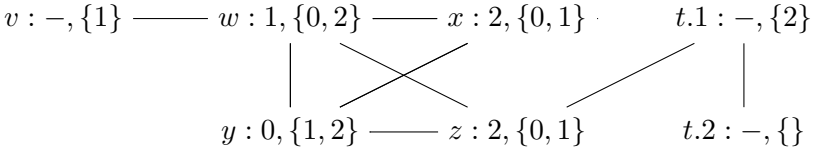

Now we repeat the process, selecting another maximally saturated node. This time there is a three-way tie between $w$, $x$, and $z$. We color $w$ with 1.



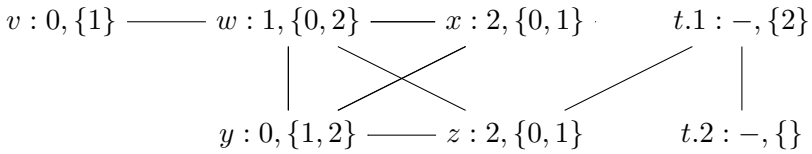

The most saturated nodes are now $x$ and $z$. We color $x$ with the next available color which is 2.



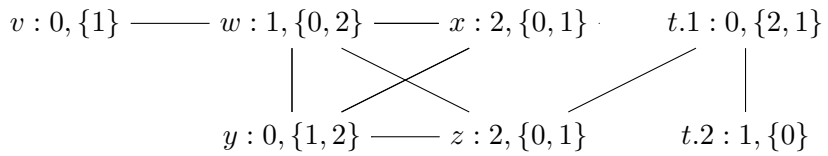

Node $z$ is the next most highly saturated, so we color $z$ with 2.




We have a 2-way tie between $v$ and $t.1$. We choose to color $v$ with 0.



In the last two steps of the algorithm, we color $t.1$ with 0 then $t.2$ with 1.




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 just have one extra register to use for register allocation, just `rbx`. Then the following is the mapping of colors to registers and stack allocations.

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

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

$$\{v \mapsto \texttt{\%rbx},\ w \mapsto \texttt{-8(\%rbp)},\ x \mapsto \texttt{-16(\%rbp)},\ y \mapsto \texttt{\%rbx},\ z \mapsto \texttt{-16(\%rbp)},$$
$$t.1 \mapsto \texttt{\%rbx},\ t.2 \mapsto \texttt{-8(\%rbp)}\}$$

Applying this assignment to our running example (Figure 3.1) yields the program on the right.

```
(program (v w x y z)                (program 16
  (movq (int 1) (var v))              (movq (int 1) (reg rbx))
  (movq (int 46) (var w))             (movq (int 46) (deref rbp -8))
  (movq (var v) (var x))              (movq (reg rbx) (deref rbp -16))
  (addq (int 7) (var x))              (addq (int 7) (deref rbp -16))
  (movq (var x) (var y))              (movq (deref rbp -16) (reg rbx))
  (addq (int 4) (var y))              (addq (int 4) (reg rbx))
  (movq (var x) (var z))        ⇒     (movq (deref rbp -16) (deref rbp -16))
  (addq (var w) (var z))              (addq (deref rbp -8) (deref rbp -16))
  (movq (var y) (var t.1))            (movq (reg rbx) (reg rbx))
  (negq (var t.1))                    (negq (reg rbx))
  (movq (var z) (var t.2))            (movq (deref rbp -16) (deref rbp -8))
  (addq (var t.1) (var t.2))          (addq (reg rbx) (deref rbp -8))
  (movq (var t.2) (reg rax)))         (movq (deref rbp -8) (reg rax)))
```

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 `-16(%rbp)` to itself is deleted and the addition of `-8(%rbp)` to `-16(%rbp)` is fixed by going through `rax`. The following shows the portion of the program that changed.

```
(addq (int 4) (reg rbx))
(movq (deref rbp -8) (reg rax)
(addq (reg rax) (deref rbp -16))
```

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

**Exercise 11.** Implement the pass `allocate-registers` and test it by creating new example programs that exercise all of the register allocation algorithm, such as forcing variables to be spilled to the stack.

I recommend organizing our code by creating a helper function named `color-graph` that takes an interference graph and a list of all the variables in the program. This function should return a mapping of variables to their colors. By creating this helper function, we will be able to reuse it in Chapter 6 when we add support for functions. Once you have obtained the coloring from `color-graph`, you can assign the variables to registers or stack locations based on their color and then use the `assign-homes` function from Section 2.7 to replace the variables with their assigned location.

## 3.4 Print x86 and Conventions for Registers

Recall the 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 there are agreed-upon conventions for how different functions share the same fixed set of registers. There is a function inside the operating system (OS) that calls our `main` function, and that OS function uses the same registers that we use in `main`. The convention for x86 is that the caller is responsible for freeing up some registers, the *caller save registers*, prior to the function call, and the callee is responsible for saving and restoring some other registers, the *callee save registers*, before and after using them. The caller save registers are

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

while the callee save 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 save registers get overwritten with arbitrary values by the callee. On the other hand, the caller can safely assume that all the callee save registers contain the same values after the call

that they did before the call. The callee can freely use any of the caller save registers. However, if the callee wants to use a callee save 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.

The upshot of these conventions is that the `main` function needs to save (in the prelude) and restore (in the conclusion) any callee save registers that get used during register allocation. The simplest approach is to save and restore all the callee save registers. The more efficient approach is to keep track of which callee save registers were used and only save and restore them. Either way, make sure to take this use of stack space into account when you round up the size of the frame to make sure it is a multiple of 16 bytes.

## 3.5 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.

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

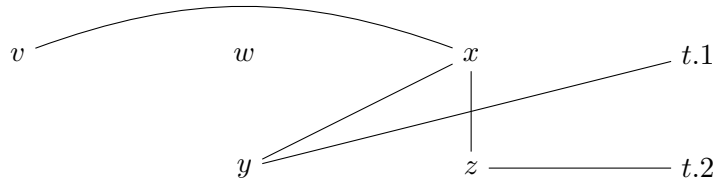
Using the same assignment that was produced by register allocator described in the last section, we get the following program.

```
(program (v w x y z)                (program 0
  (movq (int 1) (var v))              (movq (int 1) (reg rbx))
  (movq (int 46) (var w))             (movq (int 46) (reg rcx))
  (movq (var v) (var x))              (movq (reg rbx) (reg rdx))
  (addq (int 7) (var x))              (addq (int 7) (reg rdx))
  (movq (var x) (var y))              (movq (reg rdx) (reg rbx))
  (addq (int 4) (var y))              (addq (int 4) (reg rbx))
  (movq (var x) (var z))        ⇒     (movq (reg rdx) (reg rdx))
  (addq (var w) (var z))              (addq (reg rcx) (reg rdx))
  (movq (var y) (var t.1))            (movq (reg rbx) (reg rbx))
  (negq (var t.1))                    (negq (reg rbx))
  (movq (var z) (var t.2))            (movq (reg rdx) (reg rcx))
  (addq (var t.1) (var t.2))          (addq (reg rbx) (reg rcx))
  (movq (var t.2) (reg rax)))         (movq (reg rcx) (reg rax)))
```

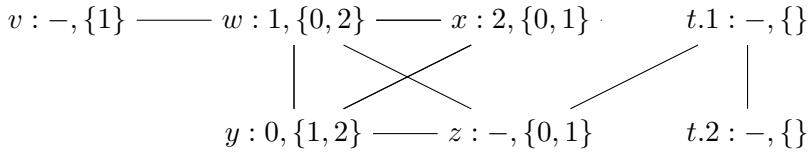
While this allocation is quite good, we could do better. For example, the variables `v` and `x` ended up in different registers, but if they had been placed in the same register, then the move from `v` to `x` could be removed.

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.

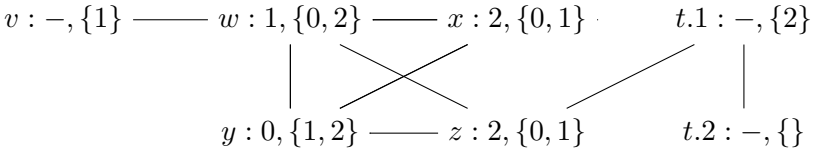
We recommend that you represent the move relationships in a graph, similar to how we represented interference. The following is the *move graph* for our running example.



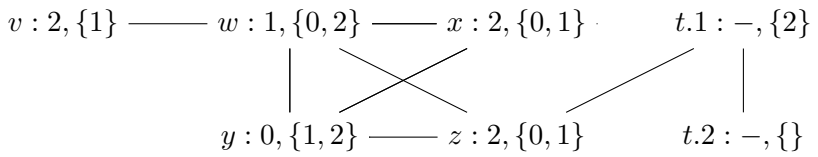

Now we replay the graph coloring, pausing to see the coloring of $z$ and $v$. So we have the following coloring so far and the most saturated vertex is $z$.




Last time we chose to color $z$ with 2, which so happens to be the color of $x$, and $z$ is move related to $x$. This was rather lucky, and if the program had been a little different, and say $x$ had been already assigned to 3, then $z$ would still get 2 and our luck would have run out. With move biasing, we use the fact that $z$ and $x$ are move related to influence the choice of color for $z$, in this case choosing 2 because that's the color of $x$.



Next we consider coloring the variable $v$, and we just need to avoid choosing 1 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 2.




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

```
(program (v w x y z)
  (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) (var t.2))
  (addq (var t.1) (var t.2))
  (movq (var t.2) (reg rax)))
```

$\Rightarrow$

```
(program 0
  (movq (int 1) (reg rdx))
  (movq (int 46) (reg rcx))
  (movq (reg rdx) (reg rdx))
  (addq (int 7) (reg rdx))
  (movq (reg rdx) (reg rbx))
  (addq (int 4) (reg rbx))
  (movq (reg rdx) (reg rdx))
  (addq (reg rcx) (reg rdx))
  (movq (reg rbx) (reg rbx))
  (negq (reg rbx))
  (movq (reg rdx) (reg rcx))
  (addq (reg rbx) (reg rcx))
  (movq (reg rcx) (reg rax)))
```

The `patch-instructions` then removes the trivial moves from `v` to `x`, from `x` to `z`, and from `y` to `t.1`, to obtain the following result.

```
(program 0
  (movq (int 1) (reg rdx))
  (movq (int 46) (reg rcx))
  (addq (int 7) (reg rdx))
  (movq (reg rdx) (reg rbx))
  (addq (int 4) (reg rbx))
  (addq (reg rcx) (reg rdx))
  (negq (reg rbx))
  (movq (reg rdx) (reg rcx))
  (addq (reg rbx) (reg rcx))
  (movq (reg rcx) (reg rax)))
```

**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.

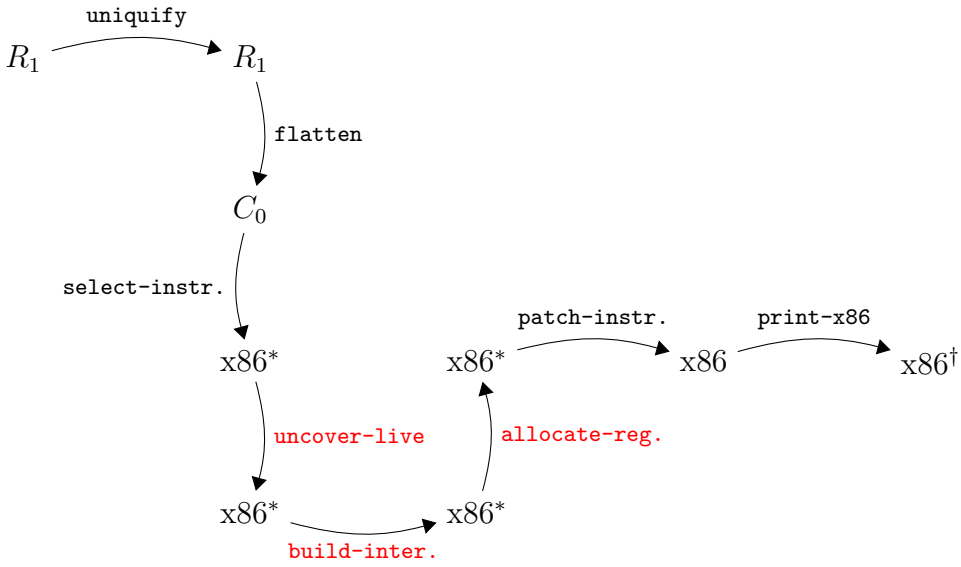



Figure 3.6: Diagram of the passes for $R_1$ with register allocation.

# 4

# Booleans, Control Flow, and Type Checking

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 the type of the argument is expected 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

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

Figure 4.1: The syntax of $R_2$, extending $R_1$ with Booleans and conditionals.

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 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 4.3. 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 the `and` and `not` on Booleans, 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 repet-

itive. 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. It is important for that match clause to come last because it matches *any* compound S-expression. 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 into 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 `typecheck-R2` 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 lookup its type in the association list.

To print the resulting value correctly, the overall type of the program must be threaded through the remainder of the passes. We can store the

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

(define (interp-op op)
  (match op
     ['+ fx+]
     ['- (lambda (n) (fx- 0 n))]
     ['not (lambda (v) (match v [#t #f] [#f #t]))]
     ['read read-fixnum]
     ['eq? (lambda (v1 v2)
             (cond [(or (and (fixnum? v1) (fixnum? v2))
                        (and (boolean? v1) (boolean? v2))
                        (and (vector? v1) (vector? 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-R2 env)
  (lambda (e)
    (define recur (interp-R2 env))
    (match e
      ...
      [(? boolean?) e]
      [`(if ,(app recur cnd) ,thn ,els)
       (match cnd
         [#t (recur thn)]
         [#f (recur els)])]
      [`(not ,(app recur v))
       (match v [#t #f] [#f #t])]
      [`(and ,(app recur v1) ,e2)
        (match v1
          [#t (match (recur e2) [#t #t] [#f #f])]
          [#f #f])]
      [`(,op ,(app recur args) ...)
        #:when (set-member? primitives op)
        (apply (interp-op op) args)]
     )))
```

Figure 4.2: Interpreter for the $R_2$ language.

```
(define (typecheck-R2 env)
  (lambda (e)
    (define recur (typecheck-R2 env e))
    (match e
      [(? fixnum?) 'Integer]
      [(? boolean?) 'Boolean]
      [(? symbol?) (lookup e env)]
      [`(read)     'Integer]
      [`(let ([,x ,(app recur T)]) ,body)
       (define new-env (cons (cons x T) env))
       (typecheck-R2 new-env body)]
      ...
      [`(not ,(app (typecheck-R2 env) T))
       (match T
         ['Boolean 'Boolean]
         [else (error 'typecheck-R2 "'not' expects a Boolean" e)])]
      ...
      [`(program ,body)
       (define ty ((typecheck-R2 '()) body))
       `(program (type ,ty) ,body)]
      )))
```

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

| | | |
|---|---|---|
| *arg* | ::= | *int* \| *var* \| `#t` \| `#f` |
| *cmp* | ::= | `eq?` \| `<` \| `<=` \| `>` \| `>=` |
| *exp* | ::= | *arg* \| (`read`) \| (`-` *arg*) \| (`+` *arg arg*) \| (`not` *arg*) \| (*cmp arg arg*) |
| *stmt* | ::= | (`assign` *var exp*) \| (`return` *arg*) |
| | \| | (`if` (*cmp arg arg*) *stmt*$^*$ *stmt*$^*$) |
| $C_1$ | ::= | (`program` (*var*$^*$) (`type` *type*) *stmt*$^+$) |

Figure 4.4: The $C_1$ language, extending $C_0$ with Booleans and conditionals.

type within the `program` form as shown in Figure 4.3. The syntax for post-typechecking $R_2$ programs as follows:

$$R_2 \quad ::= \quad (\texttt{program}\ (\texttt{type}\ \textit{type})\ \textit{exp})$$

**Exercise 13.** Complete the implementation of `typecheck-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 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 The $C_1$ Language

The $R_2$ language adds Booleans and conditional expressions to $R_1$. As with $R_1$, we shall compile to a C-like intermediate language, but we need to grow that intermediate language to handle the new features in $R_2$. Figure 4.4 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, and we add an `if` statement. The `if` statement of $C_1$ includes an `eq?` test, which is needed for improving code generation in Section 4.11. We do not include `and` in $C_1$ because it is not needed in the translation of the `and` of $R_2$.

## 4.4 Flatten Expressions

We expand the `flatten` pass to handle the Boolean literals `#t` and `#f`, the new logic and comparison operations, and `if` expressions. We shall start with a simple example of translating a `if` expression, shown below on the left.

```
(program (if #f 0 42))        ⇒      (program (if.1)
                                       (if (eq? #t #f)
                                         ((assign if.1 0))
                                         ((assign if.1 42)))
                                       (return if.1))
```

The value of the `if` expression is the value of the branch that is selected. Recall that in the `flatten` pass we need to replace arbitrary expressions with *arg*'s (variables or literals). In the translation above, on the right, we have replaced the `if` expression with a new variable `if.1`, inside `(return if.1)`, and we have produced code that will assign the appropriate value to `if.1` using an `if` statement prior to the `return`. For $R_1$, the `flatten` pass returned a list of assignment statements. Here, for $R_2$, we return a list of statements that can include both `if` statements and assignment statements.

The next example is a bit more involved, showing what happens when there are complex expressions (not variables or literals) in the condition and branch expressions of an `if`, including nested `if` expressions.

```
(program                                    (program (t.1 t.2 if.1 t.3 t.4
  (if (eq? (read) 0)                                  if.2 t.5)
      777                                     (assign t.1 (read))
      (+ 2 (if (eq? (read) 0)               (assign t.2 (eq? t.1 0))
               40                           (if (eq? #t t.2)
               444))))                        ((assign if.1 777))
                                              ((assign t.3 (read))
                                               (assign t.4 (eq? t.3 0))
                                               (if (eq? #t t.4)
                                                 ((assign if.2 40))
                                                 ((assign if.2 444)))
                                               (assign t.5 (+ 2 if.2))
                                               (assign if.1 t.5)))
                                            (return if.1))
```

The `flatten` clauses for the Boolean literals and the operations `not` and `eq?` are straightforward. However, the `flatten` clause for `and` requires some care to properly imitate the order of evaluation of the interpreter for $R_2$ (Figure 4.2). We recommend using an `if` statement in the code you

generate for `and`.

The `flatten` clause for `if` also requires some care because the condition of the `if` can be an arbitrary expression in $R_2$, but in $C_1$ the condition must be an equality predicate. For now we recommend flattening the condition into an *arg* and then comparing it with `#t`. We discuss a more efficient approach in Section 4.11.

**Exercise 14.** Expand your `flatten` pass to handle $R_2$, that is, handle the Boolean literals, the new logic and comparison operations, and the `if` expressions. Create 4 more test cases that expose whether your flattening code is correct. Test your `flatten` pass by running the output programs with `interp-C` (Appendix 12.1).

## 4.5 XOR, Comparisons, and Control Flow in x86

To implement the new logical operations, the comparison operations, and the `if` statement, we need to delve further into the x86 language. Figure 5.13 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 `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

| | | |
|---|---|---|
| *arg* | ::= | (`int` *int*) \| (`reg` *register*) \| (`deref` *register int*) |
| | \| | (`byte-reg` *register*) |
| *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*) \| (`jmp-if` *cc label*) |
| | \| | (`label` *label*) |
| $x86_1$ | ::= | (`program` *info* (`type` *type*) $instr^+$) |

Figure 4.5: The $x86_1$ language (extends $x86_0$ of Figure 2.7).

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.

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. 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`.

## 4.6 Select Instructions

The `select-instructions` pass lowers from $C_1$ to another intermediate representation suitable for conducting register allocation, that is, a language close to $x86_1$.

We can take the usual approach of encoding Booleans as integers, with

true as 1 and false as 0.

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

The `not` operation can be implemented in terms of `xorq` as we discussed at the beginning of this section.

Translating the `eq?` and the other comparison 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.

$$\begin{array}{lcl} & & \texttt{(cmpq } arg_2 \ arg_1\texttt{)} \\ \texttt{(assign } lhs \texttt{ (eq? } arg_1 \ arg_2\texttt{))} & \Rightarrow & \texttt{(set e (byte-reg al))} \\ & & \texttt{(movzbq (byte-reg al) } lhs\texttt{)} \end{array}$$

Regarding `if` statements, we recommend delaying when they are lowered until the `patch-instructions` pass. The reason is that for purposes of liveness analysis, `if` statements are easier to deal with than jump instructions.

**Exercise 15.** 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?` operator, creating some if necessary. Test the output of `select-instructions` using the `interp-x86` interpreter (Appendix 12.1).

## 4.7 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.

### 4.7.1 Liveness Analysis

The addition of `if` statements brings up an interesting issue in liveness analysis. Recall that liveness analysis works backwards through the program, for each instruction it computes the variables that are live before the instruction based on which variables are live after the instruction. Now consider the situation for `(if (eq?` $e_1$ $e_2$`)` *thns* *elss*`)`, where we know the $L_{\mathsf{after}}$ set and we need to produce the $L_{\mathsf{before}}$ set. We can recursively perform liveness analysis on the *thns* and *elss* branches, using $L_{\mathsf{after}}$ as the starting point, to obtain $L^{\mathsf{thns}}_{\mathsf{before}}$ and $L^{\mathsf{elss}}_{\mathsf{before}}$ respectively. However, we do not know, during compilation, which way the branch will go, so we do not know whether to

use $L^{\mathsf{thns}}_{\mathsf{before}}$ or $L^{\mathsf{elss}}_{\mathsf{before}}$ as the $L_{\mathsf{before}}$ for the entire `if` statement. 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 variables from the two branches to be the live set for the whole `if`, as shown below. Of course, we also need to include the variables that are read in $e_1$ and $e_2$.

$$L_{\mathsf{before}} = L^{\mathsf{thns}}_{\mathsf{before}} \cup L^{\mathsf{elss}}_{\mathsf{before}} \cup \mathit{Vars}(e_1) \cup \mathit{Vars}(e_2)$$

We need the live-after sets for all the instructions in both branches of the `if` when we build the interference graph, so I recommend storing that data in the `if` statement AST as follows:

`(if (eq?` $e_1$ $e_2$`)` *thns thn−lives elss els−lives*`)`

If you wrote 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, you need to be update them to handle the new kinds of arguments and instructions in $\text{x86}_1$.

### 4.7.2 Build Interference

Many of the new instructions, such as the logical operations, can be handled in the same way as the arithmetic instructions. Thus, if your code was already quite general, it will not need to be changed to handle the logical operations. If not, I recommend that you change your code to be more general. The `movzbq` instruction should be handled like the `movq` instruction. The `if` statement is straightforward to handle because we stored the live-after sets for the two branches in the AST node as described above. Here we just need to recursively process the two branches. The output of this pass can discard the live after sets, as they are no longer needed.

### 4.7.3 Assign Homes

The `assign-homes` function (Section 2.7) needs to be updated to handle the `if` statement, simply by recursively processing the child nodes. Hopefully your code already handles the other new instructions, but if not, you can generalize your code.

**Exercise 16.** Implement the additions to the `register-allocation` pass so that it works for $R_2$ and test your compiler using your previously created programs on the `interp-x86` interpreter (Appendix 12.1).

## 4.8 Lower Conditionals (New Pass)

In the `select-instructions` pass we decided to procrastinate in the lowering of the `if` statement, thereby making liveness analysis easier. Now we need to make up for that and turn the `if` statement into the appropriate instruction sequence. The following translation gives the general idea. If the condition is true, we need to execute the *thns* branch and otherwise we need to execute the *elss* branch. So we use `cmpq` and do a conditional jump to the *thenlabel*, choosing the condition code *cc* that is appropriate for the comparison operator *cmp*. If the condition is false, we fall through to the *elss* branch. At the end of the *elss* branch we need to take care to not fall through to the *thns* branch. So we jump to the *endlabel*. All of the labels in the generated code should be created with `gensym`.

$$
\begin{array}{l}
\texttt{(if (}\mathit{cmp}\ \mathit{arg}_1\ \mathit{arg}_2\texttt{)}\ \mathit{thns}\ \mathit{elss}\texttt{)} \Rightarrow \\
\\
\end{array}
\begin{array}{l}
\texttt{(cmpq}\ \mathit{arg}_2\ \mathit{arg}_1\texttt{)} \\
\texttt{(jmp-if}\ \mathit{cc}\ \mathit{thenlabel}\texttt{)} \\
\mathit{elss} \\
\texttt{(jmp}\ \mathit{endlabel}\texttt{)} \\
\texttt{(label}\ \mathit{thenlabel}\texttt{)} \\
\mathit{thns} \\
\texttt{(label}\ \mathit{endlabel}\texttt{)}
\end{array}
$$

**Exercise 17.** Implement the `lower-conditionals` pass. Test your compiler using your previously created programs on the `interp-x86` interpreter (Appendix 12.1).

## 4.9 Patch Instructions

There are no special restrictions on the instructions `jmp-if`, `jmp`, and `label`, but there is an unusual restriction on `cmpq`. The second argument is not allowed to be an immediate value (such as a literal integer). If you are comparing two immediates, you must insert another `movq` instruction to put the second argument in `rax`.

**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 4.6 shows a simple example program in $R_2$ translated to x86, showing the results of `flatten`, `select-instructions`, and the final x86 assembly.

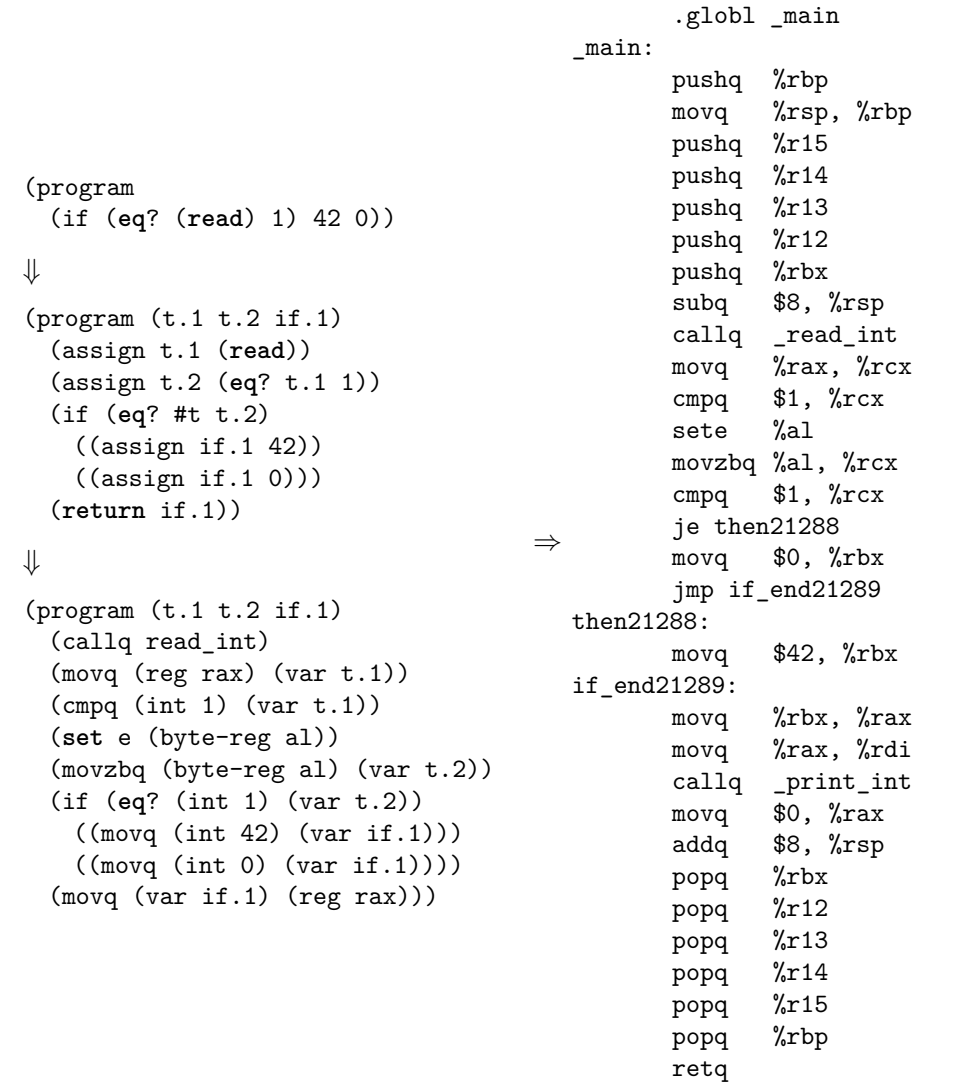



Figure 4.6: Example compilation of an `if` expression to x86.

Figure 4.7 gives an overview of all the passes needed for the compilation of $R_2$.

## 4.11 Challenge: Optimizing Conditions*

A close inspection of the x86 code generated in Figure 4.6 reveals some redundant computation regarding the condition of the `if`. We compare `rcx` to 1 twice using `cmpq` as follows.

```
cmpq   $1, %rcx
sete   %al
movzbq %al, %rcx
cmpq   $1, %rcx
je then21288
```

The reason for this non-optimal code has to do with the `flatten` pass earlier in this Chapter. We recommended flattening the condition to an *arg* and then comparing with `#t`. But if the condition is already an `eq?` test, then we would like to use that directly. In fact, for many of the expressions of Boolean type, we can generate more optimized code. For example, if the condition is `#t` or `#f`, we do not need to generate an `if` at all. If the condition is a `let`, we can optimize based on the form of its body. If the condition is a `not`, then we can flip the two branches. On the other hand, if the condition is a `and` or another `if`, we should flatten them into an *arg* to avoid code duplication.

Figure 4.8 shows an example program and the result of applying the above suggested optimizations.

**Exercise 19.** Change the `flatten` pass to improve the code that gets generated for `if` expressions. We recommend writing a helper function that recursively traverses the condition of the `if`.

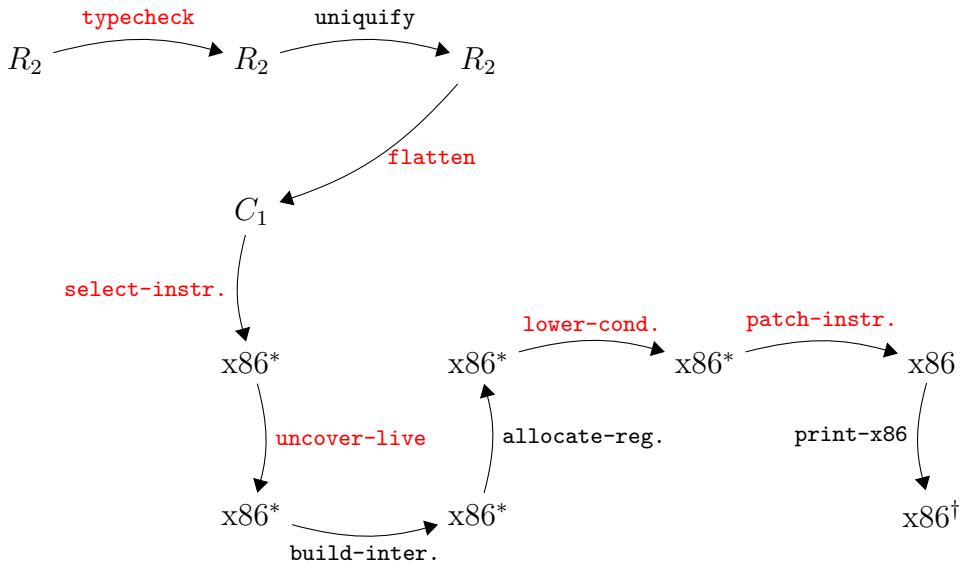



Figure 4.7: Diagram of the passes for $R_2$, a language with conditionals.

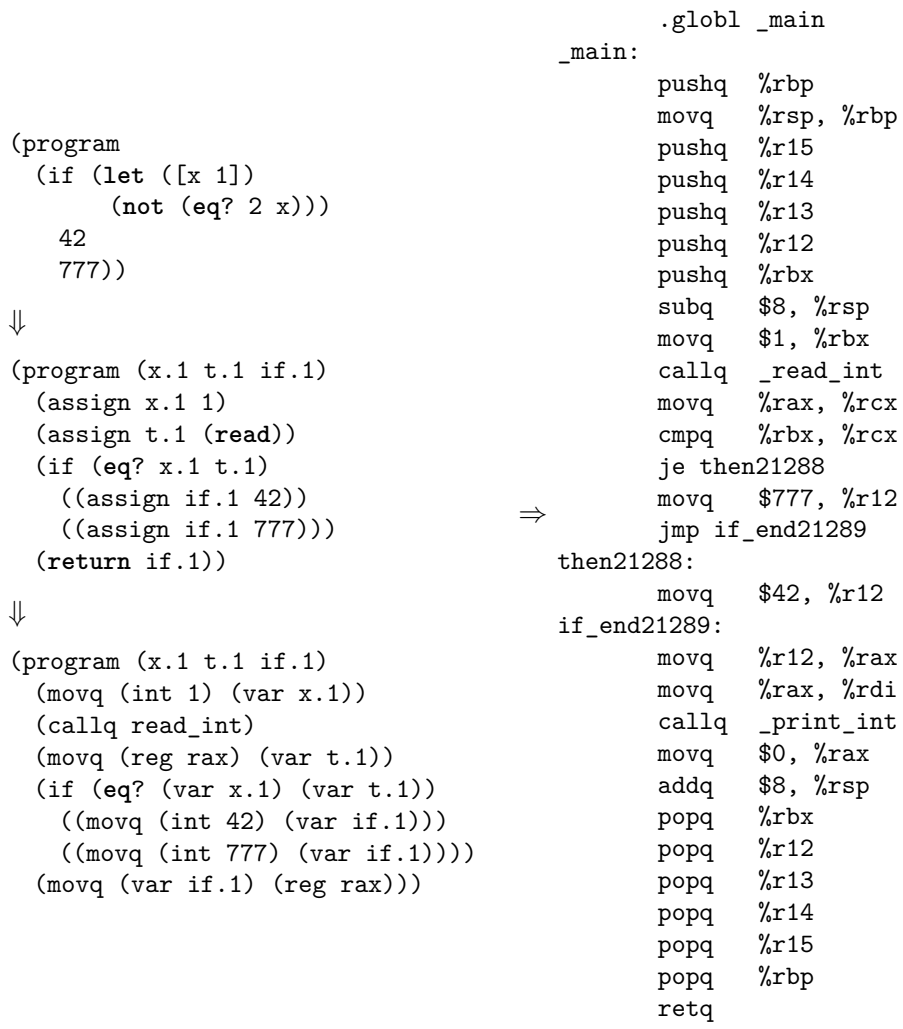



Figure 4.8: Example program with optimized conditionals.

# 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 does not follow a stack (FIFO) discipline but instead lives forever from the programmer's viewpoint. Of course, from an implementor's viewpoint, it is important to reclaim the space associated with tuples when they are 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 void values (because the `vector-set!` operation returns a void value). 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. Section 5.3 discusses all the necessary changes and additions to the compiler passes, including type checking, instruction selection, register allocation, and a new compiler pass named `expose-allocation`.

## 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.

```
(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*) |
| | \| | *var* \| (`let` ([*var exp*]) *exp*) |
| | \| | `#t` \| `#f` \| (`and` *exp exp*) \| (`not` *exp*) |
| | \| | (*cmp exp exp*) \| (`if` *exp exp exp*) |
| | \| | (`vector` *exp*$^+$) \| (`vector-ref` *exp int*) |
| | \| | (`vector-set!` *exp int exp*) |
| | \| | (`void`) |
| $R_3$ | ::= | (`program` (`type` *type*) *exp*) |

Figure 5.2: The syntax of $R_3$, extending $R_2$ with tuples.

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.[1] 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 flattening. 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 flatten pass (Section 5.3.2) this type information is propagated to all variables (including temporaries generated during flattening).

## 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 [**??????**]. In particular, we describe a two-space copying collector [**?**] that uses Cheney's algorithm to perform the copy [**?**]. 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.

[1] 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.

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. [2]

So the goal of the garbage collector is twofold:

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

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

A copying collector accomplishes this by copying all of the live objects 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

[2] 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.

```
(define (typecheck-R3 env)
  (lambda (e)
    (match e
      ...
      ['(void) (values '(has-type (void) Void) 'Void)]
      [`(vector ,(app (type-check env) e* t*) ...)
       (let ([t `(Vector ,@t*)])
         (values `(has-type (vector ,@e*) ,t) t))]
      [`(vector-ref ,(app (type-check env) e t) ,i)
       (match t
         [`(Vector ,ts ...)
          (unless (and (exact-nonnegative-integer? i)
                       (i . < . (length ts)))
                  (error 'type-check "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)])]
      [`(vector-set! ,(app (type-check env) e-vec^ t-vec) ,i
                     ,(app (type-check env) e-arg^ t-arg))
       (match t-vec
         [`(Vector ,ts ...)
          (unless (and (exact-nonnegative-integer? i)
                       (i . < . (length ts)))
            (error 'type-check "invalid index ~a" i))
          (unless (equal? (list-ref ts i) t-arg)
            (error 'type-check "type mismatch in vector-set! ~a ~a"
                   (list-ref ts i) t-arg))
          (values `(has-type (vector-set! ,e-vec^
                                          (has-type ,i Integer)
                                          ,e-arg^) Void) 'Void)]
         [else (error 'type-check
                      "expected a vector in vector-set!, not ~a" t-vec)])]
      [`(eq? ,(app (type-check env) e1 t1)
             ,(app (type-check env) e2 t2))
       (match* (t1 t2)
         [(`(Vector ,ts1 ...) `(Vector ,ts2 ...))
          (values `(has-type (eq? ,e1 ,e2) Boolean) 'Boolean)]
         [(other wise) ((super type-check env) e)])]
      )))
```

Figure 5.4: Type checker for the $R_3$ language.

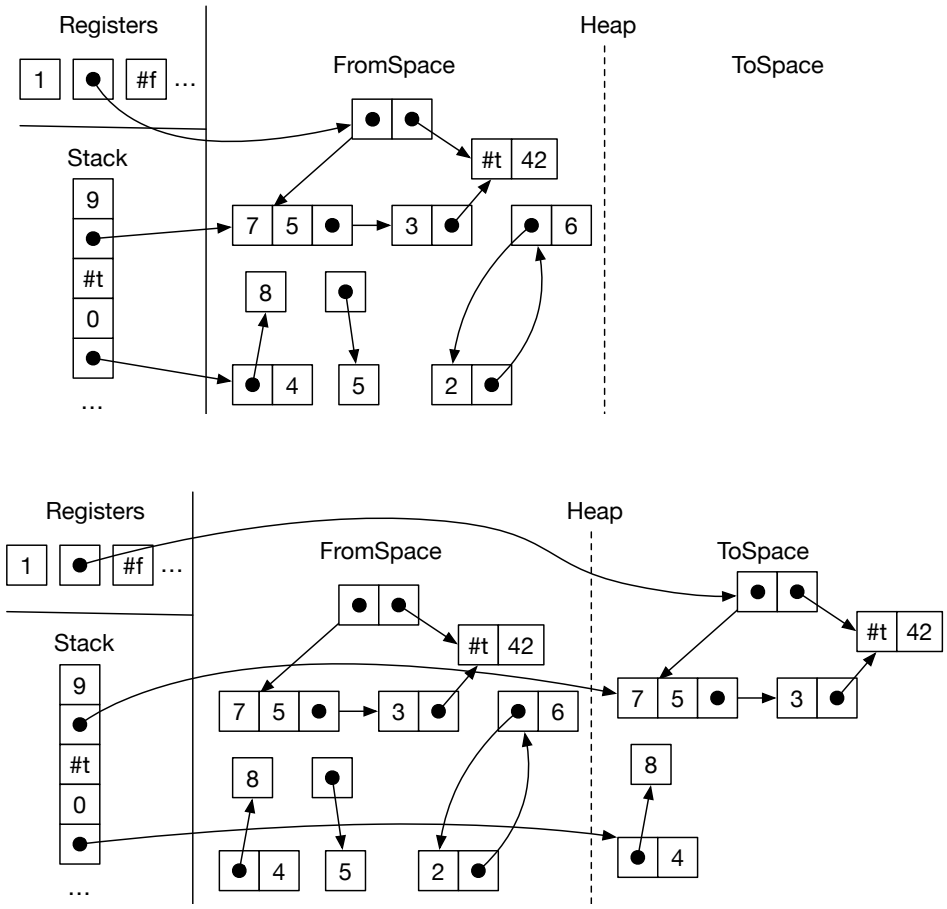



Figure 5.5: A copying collector in action.

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.

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 [**?**]. 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 [**?**]. Researchers are continuously developing new techniques and revisiting old trade-offs [**?????**].

### 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 [**??**]. Recall that such algorithms take into account the possibility of cycles by marking which vertices have already been visited, so as to ensure termination of the algorithm. These search algorithms also use a data structure such as a stack or queue as a to-do list to keep track of the vertices that need to be visited. We shall use breadth-first search and a trick due to **?** 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 [**?**].
2. Store different types of objects in different regions [**?**].
3. Use type information from the program to either generate type-specific code for collecting or to generate tables that can guide the collector [**???**].

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 [**???**], 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

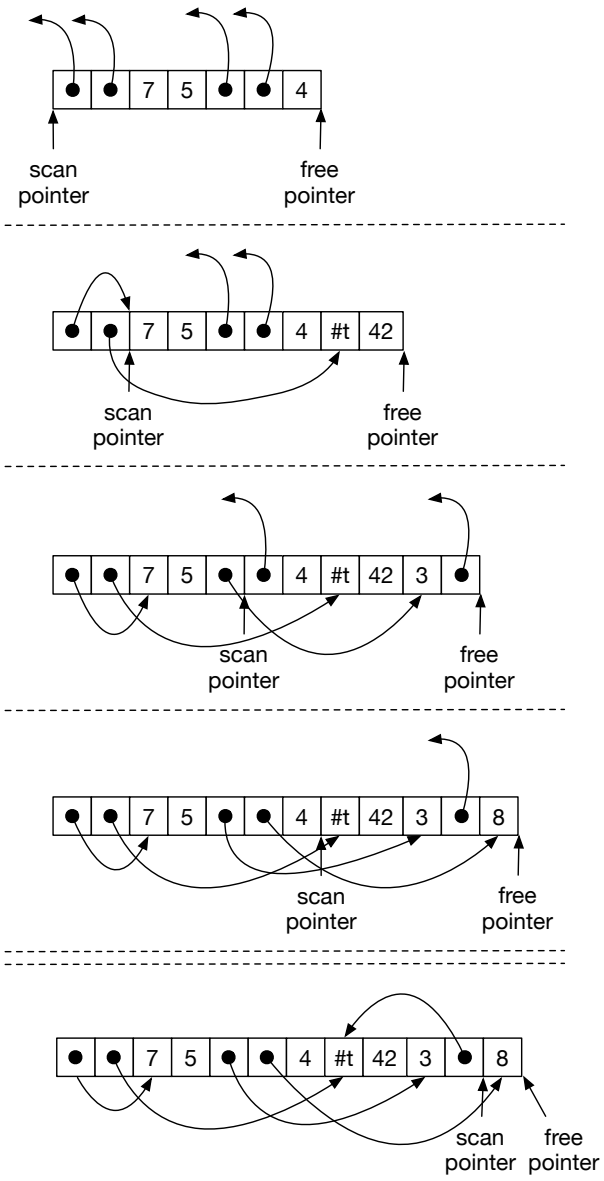



Figure 5.6: Depiction of the Cheney algorithm copying the live tuples.

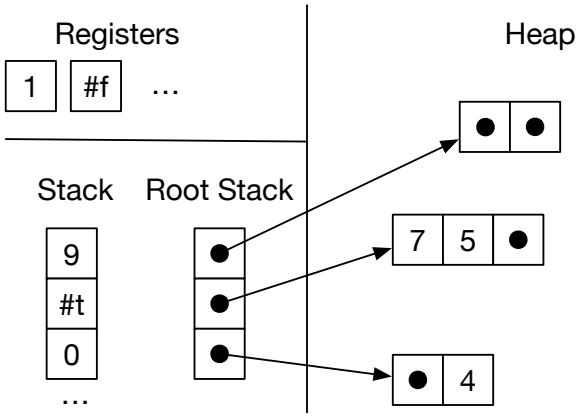



Figure 5.7: Maintaining a root stack to facilitate garbage collection.

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 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 [**?**] 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

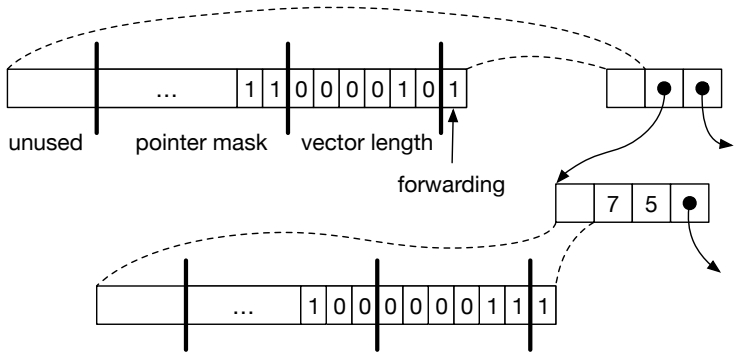



Figure 5.8: Representation for tuples in the heap.

```
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.

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 [**?**].) The `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 20.** *In the file `runtime.c` you will find the implementation of `initialize` and a partial implementation of `collect`. The `collect` func-*

*tion 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 several global variables, the* `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.*

## 5.3 Compiler Passes

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))
```

We already discuss the changes to `type-check` in Section 5.1, including the addition of `has-type`, so we proceed to discuss the new `expose-allocation` pass.

### 5.3.1 Expose Allocation (New)

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 ordering of the initializing expressions $(e_0, \ldots, e_{n-1})$ prior to the `allocate` 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* ::= $\cdots$ | (`collect` *int*) | (`allocate` *int type*) | (`global-value` *name*)

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

```
(program (type Integer)
 (vector-ref
  (vector-ref
   (let ((vecinit32990
          (let ([vecinit32986 42])
            (let ((collectret32988
                   (if (< (+ (global-value free_ptr) 16)
                          (global-value fromspace_end))
                       (void)
                       (collect 16))))
              (let ([alloc32985
                     (allocate 1 (Vector Integer))])
                (let ([initret32987
                       (vector-set! alloc32985 0 vecinit32986)])
                  alloc32985))))))
     (let ([collectret32992
            (if (< (+ (global-value free_ptr) 16)
                   (global-value fromspace_end))
                (void)
                (collect 16))])
       (let ([alloc32989 (allocate 1 (Vector (Vector Integer)))])
         (let ([initret32991 (vector-set! alloc32989 0 vecinit32990)])
           alloc32989))))
   0)
  0))
```

Figure 5.10: Output of the `expose-allocation` pass, minus all of the `has-type` 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` *arg*) |
| | \| | (`if` (*cmp arg arg*) $stmt^*$ $stmt^*$) |
| | \| | (`collect` *int*) |
| $C_2$ | ::= | (`program` ($var^*$) (`type` *type*) $stmt^+$) |

Figure 5.11: The $C_2$ language, extending $C_1$ with support for tuples.

### 5.3.2 Flatten and the $C_2$ intermediate language

The output of `flatten` 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 expressions `allocate`, `vector-ref`, and `vector-set!`, and `global-value` and the statement `collect`. The `flatten` pass can treat these new forms much like the other forms.

Recall that the `flatten` function collects all of the local variables so that it can decorate the `program` form with them. 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 `flatten` 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. For example, consider the translation of the `let` form.

```
  (let ([x (has-type rhs type)]) body)
⟹
  (values body′
          (ss1 (assign x rhs′) ss2)
          ((x . type) xt1 xt2))
```

where $rhs'$, $ss_1$, and $xs_1$ are the results of recursively flattening *rhs* and $body'$, $ss_2$, and $xs_2$ are the results of recursively flattening *body*. The output on our running example is shown in Figure 5.12.

```
'(program
  ((tmp02 . Integer) (tmp01 Vector Integer) (tmp90 Vector Integer)
   (tmp86 . Integer) (tmp88 . Void) (tmp96 . Void)
   (tmp94 . Integer) (tmp93 . Integer) (tmp95 . Integer)
   (tmp85 Vector Integer) (tmp87 . Void) (tmp92 . Void)
   (tmp00 . Void) (tmp98 . Integer) (tmp97 . Integer)
   (tmp99 . Integer) (tmp89 Vector (Vector Integer))
   (tmp91 . Void))
  (type Integer)
  (assign tmp86 42)
  (assign tmp93 (global-value free_ptr))
  (assign tmp94 (+ tmp93 16))
  (assign tmp95 (global-value fromspace_end))
  (if (< tmp94 tmp95)
    ((assign tmp96 (void)))
    ((collect 16) (assign tmp96 (void))))
  (assign tmp88 tmp96)
  (assign tmp85 (allocate 1 (Vector Integer)))
  (assign tmp87 (vector-set! tmp85 0 tmp86))
  (assign tmp90 tmp85)
  (assign tmp97 (global-value free_ptr))
  (assign tmp98 (+ tmp97 16))
  (assign tmp99 (global-value fromspace_end))
  (if (< tmp98 tmp99)
    ((assign tmp00 (void)))
    ((collect 16) (assign tmp00 (void))))
  (assign tmp92 tmp00)
  (assign tmp89 (allocate 1 (Vector (Vector Integer))))
  (assign tmp91 (vector-set! tmp89 0 tmp90))
  (assign tmp01 (vector-ref tmp89 0))
  (assign tmp02 (vector-ref tmp01 0))
  (return tmp02))
```

Figure 5.12: Output of `flatten` for the running example.

### 5.3.3 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,

| | | |
|---|---|---|
| *arg* | ::= | (`int` *int*) \| (`reg` *register*) \| (`deref` *register int*) |
| | \| | (`byte-reg` *register*) \| (`global-value` *name*) |
| *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*) |
| $x86_2$ | ::= | (`program` *info* (`type` *type*) *instr*$^+$) |

Figure 5.13: The $x86_2$ language (extends $x86_1$ of Figure 4.5).

`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 15) (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
  ((tmp02 . Integer) (tmp01 Vector Integer) (tmp90 Vector Integer)
   (tmp86 . Integer) (tmp88 . Void) (tmp96 . Void) (tmp94 . Integer)
   (tmp93 . Integer) (tmp95 . Integer) (tmp85 Vector Integer)
   (tmp87 . Void) (tmp92 . Void) (tmp00 . Void) (tmp98 . Integer)
   (tmp97 . Integer) (tmp99 . Integer) (tmp89 Vector (Vector Integer))
   (tmp91 . Void)) (type Integer)
  (movq (int 42) (var tmp86))
  (movq (global-value free_ptr) (var tmp93))
  (movq (var tmp93) (var tmp94))
  (addq (int 16) (var tmp94))
  (movq (global-value fromspace_end) (var tmp95))
  (if (< (var tmp94) (var tmp95))
    ((movq (int 0) (var tmp96)))
    ((movq (reg r15) (reg rdi))
     (movq (int 16) (reg rsi))
     (callq collect)
     (movq (int 0) (var tmp96))))
  (movq (var tmp96) (var tmp88))
  (movq (global-value free_ptr) (var tmp85))
  (addq (int 16) (global-value free_ptr))
  (movq (var tmp85) (reg r11))
  (movq (int 3) (deref r11 0))
  (movq (var tmp85) (reg r11))
  (movq (var tmp86) (deref r11 8))
  (movq (int 0) (var tmp87))
  (movq (var tmp85) (var tmp90))
  (movq (global-value free_ptr) (var tmp97))
  (movq (var tmp97) (var tmp98))
  (addq (int 16) (var tmp98))
  (movq (global-value fromspace_end) (var tmp99))
  (if (< (var tmp98) (var tmp99))
    ((movq (int 0) (var tmp00)))
    ((movq (reg r15) (reg rdi))
     (movq (int 16) (reg rsi))
     (callq collect)
     (movq (int 0) (var tmp00))))
  (movq (var tmp00) (var tmp92))
  (movq (global-value free_ptr) (var tmp89))
  (addq (int 16) (global-value free_ptr))
  (movq (var tmp89) (reg r11))
  (movq (int 131) (deref r11 0))
  (movq (var tmp89) (reg r11))
  (movq (var tmp90) (deref r11 8))
  (movq (int 0) (var tmp91))
  (movq (var tmp89) (reg r11))
  (movq (deref r11 8) (var tmp01))
  (movq (var tmp01) (reg r11))
  (movq (deref r11 8) (var tmp02))
  (movq (var tmp02) (reg rax)))
```

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

### 5.3.4 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-save registers. (They already interfere with the caller-save 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.

$x86_2 \quad ::= \quad (\texttt{program}\ (\textit{stackSpills}\ \textit{rootstackSpills})\ (\texttt{type}\ \textit{type})\ \textit{instr}^{+})$

### 5.3.5 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 $R_3$.

```
       .globl _main
_main:
       pushq  %rbp
       movq   %rsp, %rbp
       pushq  %r14
       pushq  %r13
       pushq  %r12
       pushq  %rbx
       subq   $0, %rsp
       movq $16384, %rdi
       movq $16, %rsi
       callq _initialize
       movq _rootstack_begin(%rip), %r15
       movq $0, (%r15)
       addq $8, %r15

       movq   $42, %rbx
       movq   _free_ptr(%rip), %rcx
       addq   $16, %rcx
       movq   _fromspace_end(%rip), %rdx
       cmpq   %rdx, %rcx
       jl then33131
       movq   %r15, %rdi
       movq   $16, %rsi
       callq  _collect
       movq   $0, %rcx
       jmp if_end33132
then33131:
       movq   $0, %rcx
if_end33132:
       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, %rbx
       movq   %rcx, -8(%r15)
       movq   _free_ptr(%rip), %rbx
       movq   %rbx, %rcx
       addq   $16, %rcx
       movq   _fromspace_end(%rip), %rbx
       cmpq   %rbx, %rcx
       jl then33133
       movq   %r15, %rdi
       movq   $16, %rsi
       callq  _collect
       movq   $0, %rbx
       jmp if_end33134
then33133:
       movq   $0, %rbx
if_end33134:
       movq   _free_ptr(%rip), %rbx
       addq   $16, _free_ptr(%rip)
       movq   %rbx, %r11
       movq   $131, 0(%r11)
       movq   %rbx, %r11
       movq   -8(%r15), %rax
       movq   %rax, 8(%r11)
       movq   $0, %rcx
       movq   %rbx, %r11
       movq   8(%r11), %rbx
       movq   %rbx, %r11
       movq   8(%r11), %rbx
       movq   %rbx, %rax

       movq   %rax, %rdi
       callq  _print_int
       movq   $0, %rax
       subq $8, %r15
       addq   $0, %rsp
       popq   %rbx
       popq   %r12
       popq   %r13
       popq   %r14
       popq   %rbp
       retq
```

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

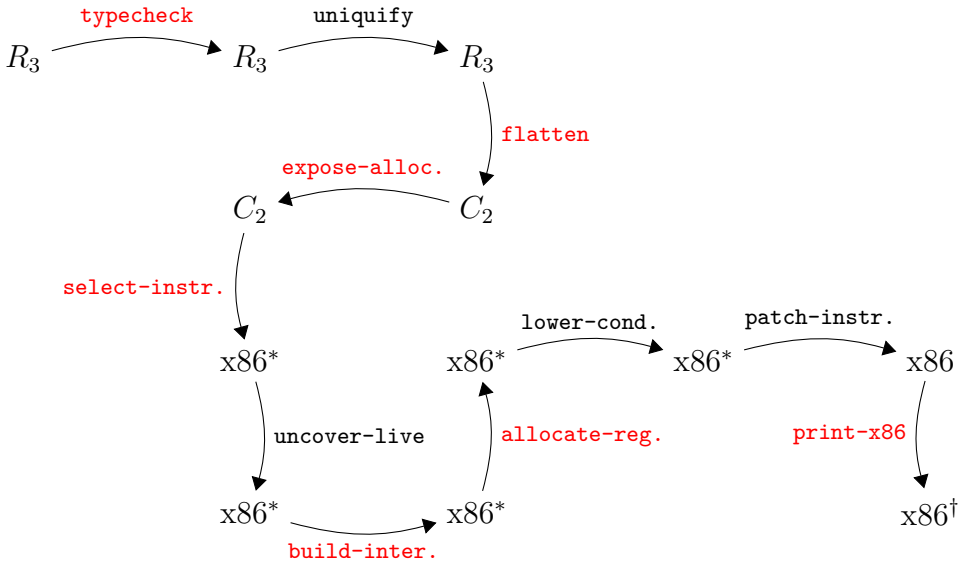



Figure 5.16: Diagram of the passes for $R_3$, a language with tuples.

# 6

# Functions

This chapter studies the compilation of functions (aka. procedures) 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. This abstraction level is an important stepping stone to implementing lexically-scoped functions in the form of `lambda` abstractions (Chapter 7).

## 6.1 The $R_4$ Language

The syntax for function definitions and function application (aka. function call) 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).

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

(*type*$_1$ $\cdots$ *type*$_n$ `->` *type*$_r$)

for a function whose $n$ parameters have the types *type*$_1$ through *type*$_n$ and whose return type is *type*$_r$. The main limitation of these functions (with respect to Racket functions) is that they are not lexically scoped. That is, the only external entities that can be referenced from inside a function body are other globally-defined functions. The syntax of $R_4$ prevents functions from being nested inside each other; they can only be defined at the top level.

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

| | | |
|---|---|---|
| *type* | ::= | `Integer` \| `Boolean` \| (`Vector` *type*$^+$) \| `Void` \| (*type*$^*$ `->` *type*) |
| *cmp* | ::= | `eq?` \| `<` \| `<=` \| `>` \| `>=` |
| *exp* | ::= | *int* \| (`read`) \| (`-` *exp*) \| (`+` *exp exp*) |
| | \| | *var* \| (`let` ([*var exp*]) *exp*) |
| | \| | `#t` \| `#f` \| (`and` *exp exp*) \| (`not` *exp*) |
| | \| | (*cmp exp exp*) \| (`if` *exp exp exp*) |
| | \| | (`vector` *exp*$^+$) \| (`vector-ref` *exp int*) |
| | \| | (`vector-set!` *exp int exp*) \| (`void`) |
| | \| | (*exp exp*$^*$) |
| *def* | ::= | (`define` (*var* [*var* `:` *type*]$^*$) `:` *type exp*) |
| $R_4$ | ::= | (`program` *def*$^*$ *exp*) |

Figure 6.1: Syntax of $R_4$, extending $R_3$ 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 $R_4$.

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.

## 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

```
(define (interp-R4 env)
  (lambda (e)
    (match e
      ....
      [`(define (,f [,xs : ,ps] ...) : ,rt ,body)
       (cons f `(lambda ,xs ,body))]
      [`(program ,ds ... ,body)
       (let ([top-level (map (interp-R4 '()) ds)])
          ((interp-R4 top-level) body))]
      [`(,fun ,args ...)
        (define arg-vals (map (interp-R4 env) args))
        (define fun-val ((interp-R4 env) fun))
        (match fun-val
           [`(lambda (,xs ...) ,body)
             (define new-env (append (map cons xs arg-vals) env))
             ((interp-R4 new-env) body)]
           [else (error "interp-R4, expected function, not" fun-val)]))]
      [else (error 'interp-R4 "unrecognized expression")]
      )))
```

Figure 6.3: Interpreter for the $R_4$ language.

`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 Sections 2.2 and 2.6 we saw the use of the `callq` instruction for jumping to a function as specified by a label. The use of the instruction changes slightly if the function is specified by an address in a register, that is, an *indirect function call*. The x86 syntax is to give the register name prefixed with an asterisk.

```
callq *%rbx
```

The x86 architecture does not directly support passing arguments to functions; instead we use a combination of registers and stack locations for passing arguments, following the conventions used by `gcc` as described by **?**. Up to six arguments may be passed in registers, using the registers `rdi`, `rsi`, `rdx`, `rcx`, `r8`, and `r9`, in that order. If there are more than six arguments, then the rest must be placed on the stack, which we call *stack arguments*, which we discuss in later paragraphs. The register `rax` is for the return value of the function.

Recall from Section 2.2 that the stack is also used for local variables and for storing the values of callee-save registers (we shall refer to all of these collectively as "locals"), and that at the beginning of a function we move the stack pointer `rsp` down to make room for them. To make additional room for passing arguments, we shall move the stack pointer even further down. We count how many stack arguments are needed for each function call that occurs inside the body of the function and find their maximum. Adding this number to the number of locals gives us how much the `rsp` should be moved at the beginning of the function. In preparation for a function call, we offset from `rsp` to set up the stack arguments. We put the first stack argument in `0(%rsp)`, the second in `8(%rsp)`, and so on.

Upon calling the function, the stack arguments are retrieved by the callee using the base pointer `rbp`. The address `16(%rbp)` is the location of the first stack argument, `24(%rbp)` is the address of the second, and so on. Figure 6.4 shows the layout of the caller and callee frames. Notice how important it is that we correctly compute the maximum number of arguments needed for function calls; if that number is too small then the arguments and local variables will smash into each other!

As discussed in Section 3.4, an x86 function is responsible for following conventions regarding the use of registers: the caller should assume that all the caller save 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 save 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 save register, the callee must arrange to put the original value back in the register prior to returning to the caller.

## 6.3 The compilation of functions

Now that we have a good understanding of functions as they appear in $R_4$ and the support for functions in x86, we need to plan the changes to our compiler, that is, do we need any new passes and/or do we need to change any existing passes? Also, do we need to add new kinds of AST nodes to any of the intermediate languages?

To begin with, the syntax of $R_4$ is inconvenient for purposes of compilation because it conflates the use of function names and local variables and it conflates the application of primitive operations and the application of functions. 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

| Caller View | Callee View | Contents | Frame |
|---|---|---|---|
| 8(%rbp) | | return address | |
| 0(%rbp) | | old rbp | |
| -8(%rbp) | | local 1 | Caller |
| ... | | ... | |
| $-8k$(%rbp) | | local $k$ | |
| | | | |
| $8n-8$(%rsp) | $8n+8$(%rbp) | argument $n$ | |
| | ... | ... | |
| 0(%rsp) | 16(%rbp) | argument 1 | |
| | 8(%rbp) | return address | |
| | 0(%rbp) | old rbp | |
| | -8(%rbp) | local 1 | Callee |
| | ... | ... | |
| | $-8m$(%rsp) | local $m$ | |

Figure 6.4: Memory layout of caller and callee frames.

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

Figure 6.5: The $F_1$ language, an extension of $R_3$ (Figure 5.2).

$$
\begin{array}{lcl}
\mathit{arg} & ::= & \mathit{int} \mid \mathit{var} \mid \texttt{\#t} \mid \texttt{\#f} \mid (\texttt{function-ref}\ \mathit{label}) \\
\mathit{cmp} & ::= & \texttt{eq?} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \\
\mathit{exp} & ::= & \mathit{arg} \mid (\texttt{read}) \mid (\texttt{-}\ \mathit{arg}) \mid (\texttt{+}\ \mathit{arg}\ \mathit{arg}) \mid (\texttt{not}\ \mathit{arg}) \mid (\mathit{cmp}\ \mathit{arg}\ \mathit{arg}) \\
 & \mid & (\texttt{vector}\ \mathit{arg}^{+}) \mid (\texttt{vector-ref}\ \mathit{arg}\ \mathit{int}) \\
 & \mid & (\texttt{vector-set!}\ \mathit{arg}\ \mathit{int}\ \mathit{arg}) \\
 & \mid & (\texttt{app}\ \mathit{arg}\ \mathit{arg}^{*}) \\
\mathit{stmt} & ::= & (\texttt{assign}\ \mathit{var}\ \mathit{exp}) \mid (\texttt{return}\ \mathit{arg}) \\
 & \mid & (\texttt{if}\ (\mathit{cmp}\ \mathit{arg}\ \mathit{arg})\ \mathit{stmt}^{*}\ \mathit{stmt}^{*}) \\
 & \mid & (\texttt{initialize}\ \mathit{int}\ \mathit{int}) \\
 & \mid & (\texttt{if}\ (\texttt{collection-needed?}\ \mathit{int})\ \mathit{stmt}^{*}\ \mathit{stmt}^{*}) \\
 & \mid & (\texttt{collect}\ \mathit{int}) \mid (\texttt{allocate}\ \mathit{int}) \\
 & \mid & (\texttt{call-live-roots}\ (\mathit{var}^{*})\ \mathit{stmt}^{*}) \\
\mathit{def} & ::= & (\texttt{define}\ (\mathit{label}\ [\mathit{var}\texttt{:}\mathit{type}]^{*})\texttt{:}\mathit{type}\ \mathit{stmt}^{+}) \\
C_3 & ::= & (\texttt{program}\ (\mathit{var}^{*})\ (\texttt{type}\ \mathit{type})\ (\texttt{defines}\ \mathit{def}^{*})\ \mathit{stmt}^{+})
\end{array}
$$

Figure 6.6: The $C_3$ language, extending $C_2$ with functions.

move the function name to a register. Similarly, the application of a function is going to require a complex sequence of instructions, unlike the primitive operations. Thus, it is a good idea to create a new pass that changes function references from just a symbol $f$ to `(function-ref` $f$`)` and that changes function application from `(`$e_0\ e_1\ \ldots\ e_n$`)` to the explicitly tagged AST `(app` $e_0\ e_1\ \ldots\ e_n$`)`. 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 `flatten` pass because `flatten` will help us compile `function-ref`. Figure 6.6 defines the syntax for $C_3$, the output of `flatten`.

Because each `function-ref` needs to eventually become an `leaq` instruction, it first needs to become an assignment statement so there is a left-hand side in which to put the result. This can be handled easily in the `flatten` pass by categorizing `function-ref` as a complex expression. Then, in the `select-instructions` pass, an assignment of `function-ref` becomes a `leaq` instruction as follows:

`(assign` *lhs* `(function-ref` *f*`))` $\Rightarrow$ `(leaq (function-ref` *f*`)` *lhs*`)`

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

| | | |
|---|---|---|
| *arg* | ::= | (`int` *int*) \| (`reg` *register*) \| (`deref` *register int*) \| (`byte-reg` *register*) |
| | \| | (`global-value` *name*) |
| *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*) \| (`leaq` *arg arg*) |
| *def* | ::= | (`define` (*label*) *int info* $stmt^+$) |
| $x86_3$ | ::= | (`program` *info* (`type` *type*) (`defines` $def^*$) $instr^+$) |

Figure 6.7: The $x86_3$ language (extends $x86_2$ of Figure 5.13).

Next we consider compiling function definitions. The `flatten` pass should handle function definitions a lot like a `program` node; after all, the `program` node represents the `main` function. So the `flatten` pass, in addition to flattening the body of the function into a sequence of statements, should record the local variables in the $var^*$ field as shown below.

(`define` (*f* [*xs* `:` *ts*]$^*$) `:` *rt* (*var*$^*$) $stmt^+$)

In the `select-instructions` pass, we need to encode the parameter passing in terms of the conventions discussed in Section 6.2. So depending on the length of the parameter list *xs*, some of them may be in registers and some of them may be on the stack. I recommend generating `movq` instructions to move the parameters from their registers and stack locations into the variables *xs*, then let register allocation handle the assignment of those variables to homes. After this pass, the *xs* can be added to the list of local variables. As mentioned in Section 6.2, we need to find out how far to move the stack pointer to ensure we have enough space for stack arguments in all the calls inside the body of this function. This pass is a good place to do this and store the result in the *maxStack* field of the output `define` shown below.

(`define` (*f*) *numParams* (*var*$^*$ *maxStack*) $instr^+$)

Next, consider the compilation of function applications, which have the following form at the start of `select-instructions`.

(`assign` *lhs* (`app` *fun args* ...))

In the mirror image of handling the parameters of function definitions, some of the arguments *args* need to be moved to the argument passing registers

and the rest should be moved to the appropriate stack locations, as discussed in Section 6.2. As you're generating the code for parameter passing, take note of how many stack arguments are needed for purposes of computing the *maxStack* discussed above.

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)
```

The rest of the passes need only minor modifications to handle the new kinds of AST nodes: `function-ref`, `indirect-callq`, and `leaq`. Inside `uncover-live`, when computing the $W$ set (written variables) for an `indirect-callq` instruction, I recommend including all the caller save registers, which will have the affect of making sure that no caller save register actually needs to be saved. In `patch-instructions`, you should deal with the x86 idiosyncrasy that the destination argument of `leaq` must be a register.

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

`(function-ref` *label*`)` $\Rightarrow$ *label*`(%rip)`
`(indirect-callq` *arg*`)` $\Rightarrow$ `callq *`*arg*

For function definitions, the `print-x86` pass should add the code for saving and restoring the callee save registers, if you haven't already done that.

## 6.4 An Example Translation

Figure 6.8 shows an example translation of a simple function in $R_4$ to x86. The figure includes the results of the `flatten` and `select-instructions` passes. Can you see any ways to improve the translation?

**Exercise 21.** 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 test your compiler on these new programs and all of your previously created test programs.

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

⇓

```
(program (t.1 t.2)
  (defines
    (define (add.1 [x.1 : Integer]
                   [y.1 : Integer])
      : Integer (t.3)
      (assign t.3 (+ x.1 y.1))
      (return t.3)))
  (assign t.1 (function-ref add.1))
  (assign t.2 (app t.1 40 2))
  (return t.2))
```

⇓

```
(program ((rs.1 t.1 t.2) 0)
  (type Integer)
  (defines
   (define (add28545) 3
           ((rs.2 x.2 y.3 t.4) 0)
     (movq (reg rdi) (var rs.2))
     (movq (reg rsi) (var x.2))
     (movq (reg rdx) (var y.3))
     (movq (var x.2) (var t.4))
     (addq (var y.3) (var t.4))
     (movq (var t.4) (reg rax))))
  (movq (int 16384) (reg rdi))
  (movq (int 16) (reg rsi))
  (callq initialize)
  (movq (global-value rootstack_begin)
        (var rs.1))
  (leaq (function-ref add28545) (var t.1))
  (movq (var rs.1) (reg rdi))
  (movq (int 40) (reg rsi))
  (movq (int 2) (reg rdx))
  (indirect-callq (var t.1))
  (movq (reg rax) (var t.2))
  (movq (var t.2) (reg rax)))
```

⇓

```
        .globl add28545
add28545:
        pushq  %rbp
        movq   %rsp, %rbp
        pushq  %r15
        pushq  %r14
        pushq  %r13
        pushq  %r12
        pushq  %rbx
        subq   $8, %rsp
        movq   %rdi, %rbx
        movq   %rsi, %rbx
        movq   %rdx, %rcx
        addq   %rcx, %rbx
        movq   %rbx, %rax
        addq   $8, %rsp
        popq   %rbx
        popq   %r12
        popq   %r13
        popq   %r14
        popq   %r15
        popq   %rbp
        retq

        .globl _main
_main:
        pushq  %rbp
        movq   %rsp, %rbp
        pushq  %r15
        pushq  %r14
        pushq  %r13
        pushq  %r12
        pushq  %rbx
        subq   $8, %rsp
        movq   $16384, %rdi
        movq   $16, %rsi
        callq  _initialize
        movq   _rootstack_begin(%rip), %rcx
        leaq   add28545(%rip), %rbx
        movq   %rcx, %rdi
        movq   $40, %rsi
        movq   $2, %rdx
        callq  *%rbx
        movq   %rax, %rbx
        movq   %rbx, %rax
        movq   %rax, %rdi
        callq  _print_int
        movq   $0, %rax
        addq   $8, %rsp
        popq   %rbx
        popq   %r12
        popq   %r13
        popq   %r14
        popq   %r15
        popq   %rbp
        retq
```

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

# 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*) |
| | \| | *var* \| (`let` ([*var exp*]) *exp*) \| `#t` \| `#f` \| (`and` *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*) |
| $R_5$ | ::= | (`program` *def*$^*$ *exp*) |

Figure 7.2: Syntax of $R_5$, extending $R_4$ 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 $R_3$).

We shall describe how to compile $R_5$ to $R_4$, replacing anonymous functions with top-level function definitions. However, our compiler must 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 22.** *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

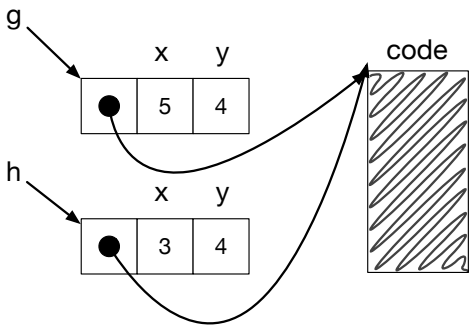



Figure 7.3: Example closure representation for the `lambda`'s in Figure 7.1.

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`. The solution is to bundle the values of the free variables together with the function pointer for the lambda's code into a data structure called a *closure*. Fortunately, we already have the appropriate ingredients to make closures, Chapter 5 gave us tuples 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 tuple. 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 code but differ in the values for free variable `x`.

## 7.2 Interpreting $R_5$

Figure 7.4 shows the definitional interpreter for $R_5$. There are several things to worth noting. First, and most importantly, the match 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`. Of course, the `lam-env` environment is extending with the mapping parameters to argument values. To enable mutual recursion and allow a unified handling of functions created with `lambda` and with `define`, the match clause for `program` includes a second pass over the top-level functions to set their environments to be the top-level environment.

```
(define (interp-R5 env)
  (lambda (ast)
    (match ast
       ...
       [`(lambda: ([,xs : ,Ts] ...) : ,rT ,body)
        `(lambda ,xs ,body ,env)]
       [`(define (,f [,xs : ,ps] ...) : ,rt ,body)
        (mcons f `(lambda ,xs ,body))]
       [`(program ,defs ... ,body)
        (let ([top-level (map (interp-R5 '()) defs)])
          (for/list ([b top-level])
                    (set-mcdr! b (match (mcdr b)
                                   [`(lambda ,xs ,body)
                                    `(lambda ,xs ,body ,top-level)])))
          ((interp-R5 top-level) body))]
       [`(,fun ,args ...)
        (define arg-vals (map (interp-R5 env) args))
        (define fun-val ((interp-R5 env) fun))
        (match fun-val
           [`(lambda (,xs ...) ,body ,lam-env)
            (define new-env (append (map cons xs arg-vals) lam-env))
            ((interp-R5 new-env) body)]
           [else (error "interp-R5, expected function, not" fun-val)])]
      )))
```

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 $R_5$.

## 7.3 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.

## 7.4 Closure Conversion

The compiling of lexically-scoped functions into C-style functions is accomplished in the pass `convert-to-closures` that comes after `reveal-functions` and before flatten. This pass needs to treat regular function calls differently from applying primitive operators, and `reveal-functions` differentiates those two cases for us.

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` (function application). 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*`)` $\Rightarrow$ `(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 : _]` *ps* `...)`

```
(let ([fvs_1 (vector-ref clos 1)])
  ...
  (let ([fvs_n (vector-ref clos n)])
    body')...))
```

The `clos` parameter refers to the closure whereas *ps* are the normal parameters of the `lambda`. The sequence of `let` forms being 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.

```
(function-ref f)  ⇒  (vector (function-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 (function-ref f) 5)))
   (let ((h (app (function-ref f) 3)))
     (+ (app g 11) (app h 15)))))
```

⇓

```
(define (f (clos.1 : _) (x : Integer)) : (Integer -> Integer)
  (let ((y 4))
    (vector (function-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 (function-ref f))))
            (app (vector-ref t.1 0) t.1 5))))
   (let ((h (let ((t.2 (vector (function-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.

# 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. (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*, and there are many kinds of polymorphism, such as subtype polymorphism [**?**] and parametric polymorphism (Chapter 10).

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)
```

Let us consider how we might compile untyped Racket to x86, thinking about the first example above. Our bit-level representation of the Boolean `#f` is zero and 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 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, creating the $R_6$ language defined in Section 8.1. Thus, $R_6$ provides the fundamental support for polymorphism and runtime types that we need to support dynamic typing.

We shall implement our untyped language $R_7$ by compiling it to $R_6$. We define $R_7$ in Section 8.2 and describe the compilation of $R_6$ and $R_7$ in the remainder of this chapter.

## 8.1 The $R_6$ Language: Typed Racket + `Any`

The syntax of $R_6$ is defined in Figure 8.1. 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 does not match $T$.

| | | |
|---|---|---|
| *type* | ::= | `Integer` \| `Boolean` \| (`Vector` *type*$^+$) \| (`Vectorof` *type*) \| `Void` |
| | \| | (*type*$^*$ `->` *type*) \| `Any` |
| *ftype* | ::= | `Integer` \| `Boolean` \| (`Vectorof Any`) \| (`Any`$^*$ `->` `Any`) |
| *cmp* | ::= | `eq?` \| `<` \| `<=` \| `>` \| `>=` |
| *exp* | ::= | *int* \| (`read`) \| (`-` *exp*) \| (`+` *exp exp*) |
| | \| | *var* \| (`let` ([*var exp*]) *exp*) |
| | \| | `#t` \| `#f` \| (`and` *exp exp*) \| (`not` *exp*) |
| | \| | (*cmp exp exp*) \| (`if` *exp exp exp*) |
| | \| | (`vector` *exp*$^+$) \| (`vector-ref` *exp int*) |
| | \| | (`vector-set!` *exp int exp*) \| (`void`) |
| | \| | (*exp exp*$^*$) \| (`lambda:` ([*var* : *type*]$^*$) : *type exp*) |
| | \| | (`inject` *exp ftype*) \| (`project` *exp ftype*) |
| | \| | (`boolean?` *exp*) \| (`integer?` *exp*) |
| | \| | (`vector?` *exp*) \| (`procedure?` *exp*) \| (`void?` *exp*) |
| *def* | ::= | (`define` (*var* [*var* : *type*]$^*$) : *type exp*) |
| $R_6$ | ::= | (`program` *def*$^*$ *exp*) |

Figure 8.1: Syntax of $R_6$, extending $R_5$ with `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. The type checker for $R_6$ is given in Figure 8.2.

Figure 8.3 shows the definitional interpreter for $R_6$.

## 8.2 The $R_7$ Language: Untyped Racket

The syntax of $R_7$, our subset of Racket, is defined in Figure 8.4. The definitional interpreter for $R_7$ is given in Figure 8.5.

## 8.3 Compiling $R_6$

Most of the compiler passes only require straightforward changes. The interesting part is in instruction selection.

```
(define type-predicates
  (set 'boolean? 'integer? 'vector? 'procedure?))

(define (typecheck-R6 env)
  (lambda (e)
    (define recur (typecheck-R6 env))
    (match e
       [`(inject ,(app recur new-e e-ty) ,ty)
        (cond
         [(equal? e-ty ty)
          (values `(inject ,new-e ,ty) 'Any)]
         [else
          (error "inject␣expected␣~a␣to␣have␣type␣~a" e ty)])]
      [`(project ,(app recur new-e e-ty) ,ty)
       (cond
        [(equal? e-ty 'Any)
         (values `(project ,new-e ,ty) ty)]
        [else
         (error "project␣expected␣~a␣to␣have␣type␣Any" e)])]
      [`(,pred ,e) #:when (set-member? type-predicates pred)
       (define-values (new-e e-ty) (recur e))
       (cond
        [(equal? e-ty 'Any)
         (values `(,pred ,new-e) 'Boolean)]
        [else
         (error "predicate␣expected␣arg␣of␣type␣Any,␣not" e-ty)])]
      [`(vector-ref ,(app recur e t) ,i)
       (match t
         [`(Vector ,ts ...) ...]
         [`(Vectorof ,t)
          (unless (exact-nonnegative-integer? i)
            (error 'type-check "invalid␣index␣~a" i))
          (values `(vector-ref ,e ,i) t)]
         [else (error "expected␣a␣vector␣in␣vector-ref,␣not" t)])]
      [`(vector-set! ,(app recur e-vec^ t-vec) ,i
                     ,(app recur e-arg^ t-arg))
       (match t-vec
         [`(Vector ,ts ...) ...]
         [`(Vectorof ,t)
          (unless (exact-nonnegative-integer? i)
            (error 'type-check "invalid␣index␣~a" i))
          (unless (equal? t t-arg)
            (error 'type-check "type␣mismatch␣in␣vector-set!␣~a␣~a"
                   t t-arg))
          (values `(vector-set! ,e-vec^
                                ,i
                                ,e-arg^) 'Void)]
         [else (error 'type-check
                      "expected␣a␣vector␣in␣vector-set!,␣not␣~a"
                      t-vec)])]
       ...
      )))
```

Figure 8.2: Type checker for 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]))]
     ...))

(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 [(equal? t1 t2)
                   v1]
                  [else
                   (error "in project, type mismatch" t1 t2)])]
           [else
            (error "in project, expected tagged value" v)])]
       ...)))
```

Figure 8.3: Interpreter for $R_6$.

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

Figure 8.4: Syntax of $R_7$, an untyped language (a subset of Racket).

```
(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 (map (interp-r7 '()) ds)])
         (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 (map recur es))]
      [`(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 8.5: Interpreter for the $R_7$ language.

**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 by the source (2) and it preserves the sign of the integer. We use the `orq` instruction to combine the tag and the value to form the tagged value.

```
(assign lhs (inject e T))   ⇒   (movq e' lhs')
                                (salq (int 2) 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 combine the value and the tag using `orq`.

```
(assign lhs (inject e T))   ⇒   (movq e' lhs')
                                (orq (int tagof(T)) lhs')
```

**Project** The instruction selection for `project` is a bit more involved. Like `inject`, the instructions 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 first check to see if the tag on the tagged value matches the tag of the target type $T$. If not, we halt the program by calling the `exit` function. If we have a match, we need to produce an untagged value by shifting it to the right by 2 bits.

```
(assign lhs (project e T))   ⇒   (movq e' lhs')
                                 (andq (int 3) lhs')
                                 (if (eq? lhs' (int tagof(T)))
                                     ((movq e' lhs')
                                      (sarq (int 2) lhs'))
                                     ((callq exit)))
```

The case for vectors and procedures begins in a similar way, checking that the runtime tag matches the target type $T$ and exiting if there is a mismatch. However, the way in which we convert the tagged value to a value is different, as there is no need to shift. Instead we need to zero-out the rightmost 2 bits. We accomplish this by creating the bit pattern $\ldots 0011$, applying `notq` to obtain $\ldots 1100$, and then applying `andq` with the tagged value get the desired result.

```
(assign lhs (project e T))    ⇒    (movq e' lhs')
                                   (andq (int 3) lhs')
                                   (if (eq? lhs' (int tagof(T)))
                                       ((movq (int 3) lhs')
                                        (notq lhs')
                                        (andq e' lhs'))
                                       ((callq exit)))
```

**Type Predicates** We leave it to the reader to devise a sequence of instructions to implement the type predicates `boolean?`, `integer?`, `vector?`, and `procedure?`.

## 8.4 Compiling $R_7$ to $R_6$

Figure 8.6 shows the compilation of many of the $R_7$ forms into $R_6$. An important invariant of this pass is that given a subexpression $e$ of $R_7$, the pass will produce an expression $e'$ of $R_6$ that has type `Any`. For example, the first row in Figure 8.6 shows the compilation of the Boolean `#t`, which must be injected to produce an expression of type `Any`. The second row of Figure 8.6, the compilation of addition, is representative of compilation for many operations: the arguments have type `Any` and must be projected to `Integer` before the addition can be performed. The compilation of `lambda` (third row of Figure 8.6) shows what happens when we need to produce type annotations, we simply use `Any`. The compilation of `if`, `eq?`, and `and` all 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. Similarly, the arguments of `and` do not need to be Boolean. For `eq?`, the arguments need not be of the same type.

| | | |
|---|---|---|
| `#t` | ⇒ | `(inject #t Boolean)` |
| `(+` $e_1$ $e_2$`)` | ⇒ | `(inject`<br>`  (+ (project` $e_1'$ `Integer)`<br>`  (project` $e_2'$ `Integer))`<br>`  Integer)` |
| `(lambda (`$x_1 \ldots$`)` $e$`)` | ⇒ | `(inject (lambda: ([`$x_1$`:Any]...):Any` $e'$`)`<br>`  (Any...Any -> Any))` |
| `(app` $e_0$ $e_1 \ldots e_n$`)` | ⇒ | `(app (project` $e_0'$ `(Any...Any -> Any))`<br>$e_1' \ldots e_n'$`)` |
| `(vector-ref` $e_1$ $e_2$`)` | ⇒ | `(let ([tmp1 (project` $e_1'$ `(Vectorof Any))])`<br>`  (let ([tmp2 (project` $e_2'$ `Integer)])`<br>`  (vector-ref tmp1 tmp2)))` |
| `(if` $e_1$ $e_2$ $e_3$`)` | ⇒ | `(if (eq?` $e_1'$ `(inject #f Boolean))`<br>$e_3'$<br>$e_2'$`)` |
| `(eq?` $e_1$ $e_2$`)` | ⇒ | `(inject (eq?` $e_1'$ $e_2'$`) Boolean)` |
| `(and` $e_1$ $e_2$`)` | ⇒ | `(let ([tmp` $e_1'$`])`<br>`  (if (eq? tmp (inject #f Boolean))`<br>`  tmp`<br>$e_2'$`))` |

Figure 8.6: Compiling $R_7$ to $R_6$.

# 9

# Gradual Typing

This chapter will be based on the ideas of **?**.

# 10

# Parametric Polymorphism

This chapter may be based on ideas from **?**, **?**, **?**, or **?**.

# 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 ...

The `map2` function ...

### 12.2.1 Graphs

- The `make-graph` function takes a list of vertices (symbols) and returns a graph.

- The `add-edge` function takes a graph and two vertices and adds an edge to the graph that connects the two vertices. The graph is updated in-place. There is no return value for this function.
- The `adjacent` function takes a graph and a vertex and returns the set of vertices that are adjacent to the given vertex. The return value is a Racket `hash-set` so it can be used with functions from the `racket/set` module.
- The `vertices` function takes a graph and returns the list of vertices in the graph.

### 12.2.2 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 flag |
| `je` $L$<br>`jl` $L$<br>`jle` $L$<br>`jg` $L$<br>`jge` $L$ | Jump to label $L$ if the flag matches the condition code, 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". |
| `jmp` $L$ | Jump to label $L$ |
| `movq` $A$, $B$ | $A \rightarrow B$ |
| `movzbq` $A$, $B$ | $A \rightarrow B$, where $A$ is a single-byte register (e.g., `al` or `cl`), $B$ is a 8-byte register, and the extra bytes of $B$ are set to zero. |
| `notq` $A$ | $\sim A \rightarrow A$ (bitwise complement) |
| `orq` $A$, $B$ | $A \mid B \rightarrow B$ (bitwise-or) |
| `andq` $A$, $B$ | $A \& B \rightarrow B$ (bitwise-and) |
| `salq` $A$, $B$ | $B \ll A \rightarrow B$ (arithmetic shift left, where $A$ is a constant) |
| `sarq` $A$, $B$ | $B \gg A \rightarrow B$ (arithmetic shift right, where $A$ is a constant) |
| `sete` $A$<br>`setl` $A$<br>`setle` $A$<br>`setg` $A$<br>`setge` $A$ | If the flag matches the condition code, then $1 \rightarrow A$, else $0 \rightarrow A$. Refer to `je` above for the description of the condition codes. $A$ must be a single byte register (e.g., `al` or `cl`). |

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

# Bibliography

Oscar Waddell and R. Kent Dybvig. Fast and effective procedure inlining. In *International Symposium on Static Analysis*, pages 35–52, september 1997. URL `http://www.cs.indiana.edu/~dyb/pubs/inlining-abstract.html`.