

Essentials of Compilation
An Incremental Approach in Racket

Essentials of Compilation
An Incremental Approach in Racket

Jeremy G. Siek

The MIT Press
Cambridge, Massachusetts
London, England

© 2021 Jeremy G. Siek. Available for free viewing or personal downloading under the CC-BY-NC-ND license.

Copyright in this monograph has been licensed exclusively to The MIT Press, <http://mitpress.mit.edu>, which will be releasing the final version to the public in 2022. All inquiries regarding rights should be addressed to The MIT Press, Rights and Permissions Department.

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

Contents

	Preface	ix
1	Preliminaries	1
	1.1 Abstract Syntax Trees	1
	1.2 Grammars	3
	1.3 Pattern Matching	5
	1.4 Recursive Functions	6
	1.5 Interpreters	6
	1.6 Example Compiler: a Partial Evaluator	9
2	Integers and Variables	11
	2.1 The \mathcal{L}_{Var} Language	11
	2.2 The x86_{Int} Assembly Language	16
	2.3 Planning the trip to x86	19
	2.4 Uniquify Variables	22
	2.5 Remove Complex Operands	24
	2.6 Explicate Control	26
	2.7 Select Instructions	28
	2.8 Assign Homes	29
	2.9 Patch Instructions	30
	2.10 Generate Prelude and Conclusion	31
	2.11 Challenge: Partial Evaluator for \mathcal{L}_{Var}	31
3	Register Allocation	33
	3.1 Registers and Calling Conventions	34
	3.2 Liveness Analysis	36
	3.3 Build the Interference Graph	39
	3.4 Graph Coloring via Sudoku	41
	3.5 Patch Instructions	47
	3.6 Prelude and Conclusion	47
	3.7 Challenge: Move Biasing	49
	3.8 Further Reading	52
4	Booleans and Conditionals	55

4.1	The \mathcal{L}_{If} Language	56
4.2	Type Checking \mathcal{L}_{If} Programs	57
4.3	The \mathcal{C}_{If} Intermediate Language	62
4.4	The x86_{If} Language	62
4.5	Shrink the \mathcal{L}_{If} Language	64
4.6	Uniquify Variables	65
4.7	Remove Complex Operands	65
4.8	Explicate Control	66
4.9	Select Instructions	72
4.10	Register Allocation	73
4.11	Patch Instructions	74
4.12	Challenge: Optimize Blocks and Remove Jumps	75
4.13	Further Reading	78
5	Loops and Dataflow Analysis	79
5.1	The $\mathcal{L}_{\text{While}}$ Language	80
5.2	Cyclic Control Flow and Dataflow Analysis	81
5.3	Mutable Variables & Remove Complex Operands	86
5.4	Uncover get!	87
5.5	Remove Complex Operands	88
5.6	Explicate Control and \mathcal{C}_{\odot}	89
5.7	Select Instructions	90
5.8	Register Allocation	90
6	Tuples and Garbage Collection	93
6.1	The $\mathcal{L}_{\text{Tuple}}$ Language	93
6.2	Garbage Collection	96
6.3	Shrink	103
6.4	Expose Allocation	104
6.5	Remove Complex Operands	105
6.6	Explicate Control and the $\mathcal{C}_{\text{Tuple}}$ language	106
6.7	Select Instructions and the $\text{x86}_{\text{Global}}$ Language	106
6.8	Register Allocation	111
6.9	Prelude and Conclusion	111
6.10	Challenge: Simple Structures	113
6.11	Challenge: Arrays	115
6.12	Challenge: Generational Collection	119
6.13	Further Reading	120
7	Functions	121
7.1	The \mathcal{L}_{Fun} Language	121
7.2	Functions in x86	126
7.3	Shrink \mathcal{L}_{Fun}	129
7.4	Reveal Functions and the $\mathcal{L}_{\text{FunRef}}$ language	129
7.5	Limit Functions	129

7.6	Remove Complex Operands	130
7.7	Explicate Control and the \mathcal{C}_{Fun} language	130
7.8	Select Instructions and the $\text{x86}_{\text{callq*}}$ Language	132
7.9	Register Allocation	134
7.10	Patch Instructions	135
7.11	Prelude and Conclusion	135
7.12	An Example Translation	136
8	Lexically Scoped Functions	139
8.1	The \mathcal{L}_λ Language	141
8.2	Assignment and Lexically Scoped Functions	144
8.3	Assignment Conversion	144
8.4	Closure Conversion	146
8.5	An Example Translation	148
8.6	Expose Allocation	149
8.7	Explicate Control and $\mathcal{C}_{\text{Clos}}$	149
8.8	Select Instructions	150
8.9	Challenge: Optimize Closures	152
8.10	Further Reading	154
9	Dynamic Typing	155
9.1	The \mathcal{L}_{Dyn} Language	155
9.2	Representation of Tagged Values	160
9.3	The \mathcal{L}_{Any} Language	160
9.4	Cast Insertion: Compiling \mathcal{L}_{Dyn} to \mathcal{L}_{Any}	166
9.5	Reveal Casts	167
9.6	Remove Complex Operands	168
9.7	Explicate Control and \mathcal{C}_{Any}	168
9.8	Select Instructions	168
9.9	Register Allocation for \mathcal{L}_{Any}	171
10	Gradual Typing	173
10.1	Type Checking $\mathcal{L}_?$ and $\mathcal{L}_{\text{Cast}}$	173
10.2	Interpreting $\mathcal{L}_{\text{Cast}}$	181
10.3	Lower Casts	182
10.4	Differentiate Proxies	185
10.5	Reveal Casts	187
10.6	Closure Conversion	188
10.7	Explicate Control	188
10.8	Select Instructions	188
10.9	Further Reading	189
11	Parametric Polymorphism	193
11.1	Compiling Polymorphism	200
11.2	Erase Types	201

A	Appendix	205
	A.1 Interpreters	205
	A.2 Utility Functions	205
	A.3 x86 Instruction Set Quick-Reference	206
	References	208
	Author Index	217
	Subject Index	219

Preface

There is a magical moment when a programmer presses the “run” button and the software begins to execute. Somehow a program written in a high-level language is running on a computer that is only capable of shuffling bits. Here we reveal the wizardry that makes that moment possible. Beginning with the groundbreaking work of Backus and colleagues in the 1950s, computer scientists discovered techniques for constructing programs, called *compilers*, that automatically translate high-level programs into machine code.

We take you on a journey of constructing your own compiler for a small but powerful language. Along the way we explain the essential concepts, algorithms, and data structures that underlie compilers. We develop your understanding of how programs are mapped onto computer hardware, which is helpful when reasoning about properties at the junction between hardware and software such as execution time, software errors, and security vulnerabilities. For those interested in pursuing compiler construction as a career, our goal is to provide a stepping-stone to advanced topics such as just-in-time compilation, program analysis, and program optimization. For those interested in designing and implementing programming languages, we connect language design choices to their impact on the compiler and the generated code.

A compiler is typically organized as a sequence of stages that progressively translate a program to the code that runs on hardware. We take this approach to the extreme by partitioning our compiler into a large number of *nanopasses*, each of which performs a single task. This enables the testing of each pass in isolation and focuses our attention, making the compiler far easier to understand.

The most familiar approach to describing compilers is with each chapter dedicated to one pass. The problem with that approach is it obfuscates how language features motivate design choices in a compiler. We instead take an *incremental* approach in which we build a complete compiler in each chapter, starting with a small input language that includes only arithmetic and variables. We add new language features in subsequent chapters, extending the compiler as necessary.

Our choice of language features is designed to elicit fundamental concepts and algorithms used in compilers.

- We begin with integer arithmetic and local variables in Chapters 1 and 2, where we introduce the fundamental tools of compiler construction: *abstract syntax trees* and *recursive functions*.
- In Chapter 3 we apply *graph coloring* to assign variables to machine registers.
- Chapter 4 adds conditional expressions, which motivates an elegant recursive algorithm for translating them into conditional `goto`'s.
- Chapter 5 adds loops and mutable variables. This elicits the need for *dataflow analysis* in the register allocator.
- Chapter 6 adds heap-allocated tuples, motivating *garbage collection*.
- Chapter 7 adds functions as first-class values but without lexical scoping, similar to functions in the C programming language (Kernighan and Ritchie 1988). The reader learns about the procedure call stack and *calling conventions* and how they interact with register allocation and garbage collection. The chapter also describes how to generate efficient tail calls.
- Chapter 8 adds anonymous functions with lexical scoping, i.e., *lambda* expressions. The reader learns about *closure conversion*, in which lambdas are translated into a combination of functions and tuples.
- Chapter 9 adds *dynamic typing*. Prior to this point the input languages are statically typed. The reader extends the statically typed language with an **Any** type which serves as a target for compiling the dynamically typed language.
- Chapter 10 uses the **Any** type of Chapter 9 to implement a *gradually typed language* in which different regions of a program may be static or dynamically typed. The reader implements runtime support for *proxies* that allow values to safely move between regions.
- Chapter 11 adds *generics* with autoboxing, leveraging the **Any** type and type casts developed in Chapters 9 and 10.

There are many language features that we do not include. Our choices balance the incidental complexity of a feature versus the fundamental concepts that it exposes. For example, we include tuples and not records because they both elicit the study of heap allocation and garbage collection but records come with more incidental complexity.

Since 2009 drafts of this book have served as the textbook for 16-week compiler courses for upper-level undergraduates and first-year graduate students at the University of Colorado and Indiana University. Students come into the course having learned the basics of programming, data structures and algorithms, and discrete mathematics. At the beginning of the course, students form groups of 2-4 people. The groups complete one chapter every two weeks, starting with Chapter 2 and finishing with Chapter 8. Many chapters include a challenge problem that we assign to the graduate students. The last two weeks of the course involve a final project in which students design and implement a compiler extension of their choosing. The later chapters can be used in support of these projects. For compiler courses at universities on the quarter system (about 10 weeks in length), we recommend completing up through Chapter 6 or Chapter 7 and providing some scaffolding code to

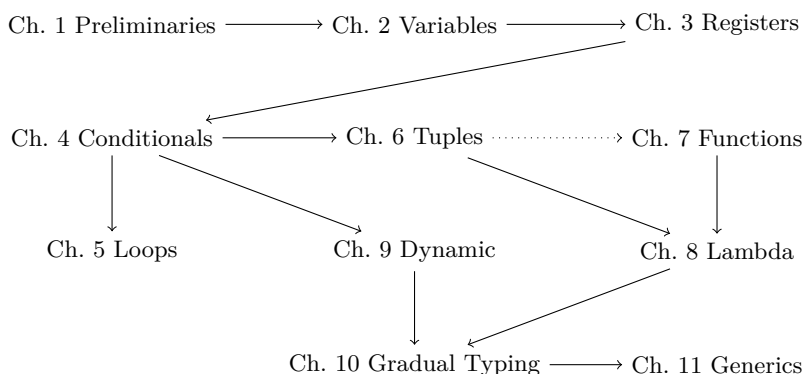
**Figure 0.1**

Diagram of chapter dependencies.

the students for each compiler pass. The course can be adapted to emphasize functional languages by skipping Chapter 5 (loops) and including Chapter 8 (lambda). The course can be adapted to dynamically typed languages by including Chapter 9. Figure 0.1 depicts the dependencies between chapters. Chapter 7 (functions) depends on Chapter 6 (tuples) only in the implementation of efficient tail calls.

This book has been used in compiler courses at California Polytechnic State University, Portland State University, Rose–Hulman Institute of Technology, University of Freiburg, University of Massachusetts Lowell, and the University of Vermont.

We use the Racket language both for the implementation of the compiler and for the input language, so the reader should be proficient with Racket or Scheme. There are many excellent resources for learning Scheme and Racket (Dybvig 1987a; Abelson and Sussman 1996; Friedman and Felleisen 1996; Felleisen et al. 2001; Felleisen et al. 2013; Flatt, Findler, and PLT 2014). The support code for this book is in the github repository at the following location:

<https://github.com/IUCompilerCourse/public-student-support-code>

The compiler targets x86 assembly language (Intel 2015), so it is helpful but not necessary for the reader to have taken a computer systems course (Bryant and O’Hallaron 2010). We introduce the parts of x86-64 assembly language that are needed in the compiler. We follow the System V calling conventions (Bryant and O’Hallaron 2005; Matz et al. 2013), so the assembly code that we generate works with the runtime system (written in C) when it is compiled using the GNU C compiler (`gcc`) on Linux and MacOS operating systems on Intel hardware. On the Windows operating system, `gcc` uses the Microsoft x64 calling convention (Microsoft 2018, 2020). So the assembly code that we generate does *not* work with the runtime system on Windows. One workaround is to use a virtual machine with Linux as the guest operating system.

Acknowledgments

The tradition of compiler construction at Indiana University goes back to research and courses on programming languages by Daniel Friedman in the 1970's and 1980's. One of his students, Kent Dybvig, implemented Chez Scheme (Dybvig 2006), an efficient, production-quality compiler for Scheme. Throughout the 1990's and 2000's, Dybvig taught the compiler course and continued the development of Chez Scheme. The compiler course evolved to incorporate novel pedagogical ideas while also including elements of real-world compilers. One of Friedman's ideas was to split the compiler into many small passes. Another idea, called "the game", was to test the code generated by each pass using interpreters.

Dybvig, with help from his students Dipanwita Sarkar and Andrew Keep, developed infrastructure to support this approach and evolved the course to use even smaller nanopasses (Sarkar, Waddell, and Dybvig 2004; Keep 2012). Many of the compiler design decisions in this book are inspired by the assignment descriptions of Dybvig and Keep (2010). In the mid 2000's a student of Dybvig's named Abdulaziz Ghuloum observed that the front-to-back organization of the course made it difficult for students to understand the rationale for the compiler design. Ghuloum proposed the incremental approach (Ghuloum 2006) that this book is based on.

We thank the many students who served as teaching assistants for the compiler course at IU including Carl Factora, Ryan Scott, Cameron Swords, and Chris Wailes. We thank Andre Kuhlenschmidt for work on the garbage collector and x86 interpreter, Michael Vollmer for work on efficient tail calls, and Michael Vitousek for help with the first offering of the incremental compiler course at IU.

We thank professors Bor-Yuh Chang, John Clements, Jay McCarthy, Joseph Near, Ryan Newton, Nate Nystrom, Peter Thiemann, Andrew Tolmach, and Michael Wollowski for teaching courses based on drafts of this book and for their feedback.

We thank Ronald Garcia for helping Jeremy survive Dybvig's compiler course in the early 2000's and especially for finding the bug that sent our garbage collector on a wild goose chase!

Jeremy G. Siek
Bloomington, Indiana

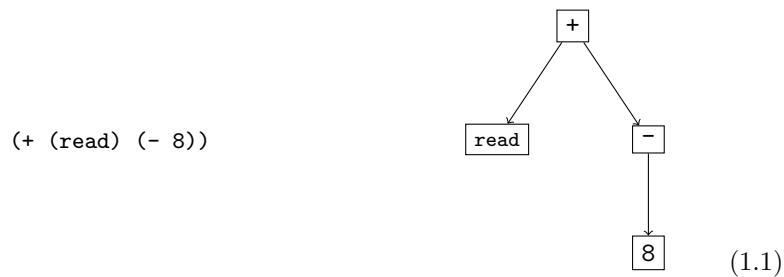
1 Preliminaries

In this chapter we review the basic tools that are needed to implement a compiler. Programs are typically input by a programmer as text, i.e., a sequence of characters. The program-as-text representation is called *concrete syntax*. We use concrete syntax to concisely write down and talk about programs. Inside the compiler, we use *abstract syntax trees* (ASTs) to represent programs in a way that efficiently supports the operations that the compiler needs to perform. The translation from concrete syntax to abstract syntax is a process called *parsing* (Aho et al. 2006). We do not cover the theory and implementation of parsing in this book. A parser is provided in the support code for translating from concrete to abstract syntax.

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

1.1 Abstract Syntax Trees

Compilers use abstract syntax trees to represent programs because they often need to ask questions like: for a given part of a program, what kind of language feature is it? What are its sub-parts? Consider the program on the left and its AST on the right. This program is an addition operation and it has two sub-parts, a read operation and a negation. The negation has another sub-part, the integer constant 8. By using a tree to represent the program, we can easily follow the links to go from one part of a program to its sub-parts.



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

We define a Racket **struct** for each kind of node. For this chapter we require just two kinds of nodes: one for integer constants and one for primitive operations. The following is the **struct** definition for integer constants.¹

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

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

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

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

The following is the **struct** definition for primitive operations.

```
(struct Prim (op args))
```

A primitive operation node includes an operator symbol **op** and a list of child **args**. For example, to create an AST that negates the number 8, we write the following.

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

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

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

The addition operator has two children:

```
(define ast1_1 (Prim '+ (list rd neg-eight)))
```

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

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

1. All of the AST structures are defined in the file `utilities.rkt` in the support code.

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

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

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

1.2 Grammars

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

The first grammar rule for the abstract syntax of \mathcal{L}_{Int} says that an instance of the `Int` structure is an expression:

$$\text{exp} ::= (\text{Int } \text{int}) \quad (1.2)$$

Each rule has a left-hand-side and a right-hand-side. If you have an AST node that matches the right-hand-side, then you can categorize it according to the left-hand-side. Symbols in typewriter font are *terminal* symbols and must literally appear in the program for the rule to be applicable. Our grammars do not mention *white-space*, that is, separating characters like spaces, tabulators, and newlines. White-space may be inserted between symbols for disambiguation and to improve readability. A name such as *exp* that is defined by the grammar rules is a *non-terminal*. The name *int* is also a non-terminal, but instead of defining it with a grammar rule, we define it with the following explanation. An *int* is a sequence of decimals (0 to 9), possibly starting with `-` (for negative integers), such that the sequence of decimals represent an integer in range -2^{62} to $2^{62} - 1$. This enables the representation of integers using 63 bits, which simplifies several aspects of compilation. Thus, these integers correspond to the Racket `fixnum` datatype on a 64-bit machine.

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

$$\text{exp} ::= (\text{Prim } \text{'read } ()) \quad (1.3)$$

The third rule categorizes the negation of an *exp* node as an *exp*.

$$\text{exp} ::= (\text{Prim } '- (\text{exp})) \quad (1.4)$$

We can apply these rules to categorize the ASTs that are in the \mathcal{L}_{Int} language. For example, by rule (1.2) `(Int 8)` is an *exp*, then by rule (1.4) the following AST is an *exp*.



The next grammar rules are for addition and subtraction expressions:

$$\text{exp} ::= (\text{Prim } '+ (\text{exp } \text{exp})) \quad (1.6)$$

$$\text{exp} ::= (\text{Prim } '- (\text{exp } \text{exp})) \quad (1.7)$$

We can now justify that the AST (1.1) is an *exp* in \mathcal{L}_{Int} . We know that `(Prim 'read ())` is an *exp* by rule (1.3) and we have already categorized `(Prim '- ((Int 8)))` as an *exp*, so we apply rule (1.6) to show that

$$(\text{Prim } '+ ((\text{Prim } '\text{read } ()) (\text{Prim } '- ((\text{Int } 8))))))$$

is an *exp* in the \mathcal{L}_{Int} language.

If you have an AST for which the above rules do not apply, then the AST is not in \mathcal{L}_{Int} . For example, the program `(* (read) 8)` is not in \mathcal{L}_{Int} because there is no rule for the `*` operator. Whenever we define a language with a grammar, the language only includes those programs that are justified by the grammar rules.

The last grammar rule for \mathcal{L}_{Int} states that there is a **Program** node to mark the top of the whole program:

$$\mathcal{L}_{\text{Int}} ::= (\text{Program } '() \text{ exp})$$

The **Program** structure is defined as follows

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

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

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

We collect all of the grammar rules for the abstract syntax of \mathcal{L}_{Int} in Figure 1.2. The concrete syntax for \mathcal{L}_{Int} is defined in Figure 1.1.

The `read-program` function provided in `utilities.rkt` of the support code reads a program in from a file (the sequence of characters in the concrete syntax of Racket) and parses it into an abstract syntax tree. See the description of `read-program` in Appendix A.2 for more details.


```

type ::= Integer
exp  ::= int | (read) | (- exp) | (+ exp exp) | (- exp exp)
 $\mathcal{L}_{\text{Int}}$  ::= exp

```

Figure 1.1

The concrete syntax of \mathcal{L}_{Int} .

```

type ::= Integer
exp  ::= (Int int) | (Prim 'read ())
        | (Prim '- (exp)) | (Prim '+ (exp exp)) | (Prim '- (exp exp))
 $\mathcal{L}_{\text{Int}}$  ::= (Program '() exp)

```

Figure 1.2

The abstract syntax of \mathcal{L}_{Int} .

1.3 Pattern Matching

As mentioned in Section 1.1, compilers often need to access the parts of an AST node. Racket provides the `match` feature to access the parts of a value. Consider the following example.

```

(match ast1_1
  [(Prim op (list child1 child2))
   (print op)])

```

In the above example, the `match` form checks whether the AST (1.1) is a binary operator and binds its parts to the three pattern variables `op`, `child1`, and `child2`. In general, a match clause consists of a *pattern* and a *body*. Patterns are recursively defined to be either a pattern variable, a structure name followed by a pattern for each of the structure's arguments, or an S-expression (symbols, lists, etc.). (See Chapter 12 of The Racket Guide² and Chapter 9 of The Racket Reference³ for complete descriptions of `match`.) The body of a match clause may contain arbitrary Racket code. The pattern variables can be used in the scope of the body, such as `op` in `(print op)`.

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

2. <https://docs.racket-lang.org/guide/match.html>

3. <https://docs.racket-lang.org/reference/match.html>

<pre> (define (leaf arith) (match arith [(Int n) #t] [(Prim 'read '()) #t] [(Prim '- (list e1)) #f] [(Prim '+ (list e1 e2)) #f] [(Prim '- (list e1 e2)) #f])) (leaf (Prim 'read '())) (leaf (Prim '- (list (Int 8)))) (leaf (Int 8)) </pre>	<pre> #t #f #t </pre>
--	-----------------------

When constructing a `match` expression, we refer to the grammar definition to identify which non-terminal we are expecting to match against, then we make sure that 1) we have one clause for each alternative of that non-terminal and 2) that the pattern in each clause corresponds to the corresponding right-hand side of a grammar rule. For the `match` in the `leaf` function, we refer to the grammar for \mathcal{L}_{Int} in Figure 1.2. The *exp* non-terminal has 4 alternatives, so the `match` has 4 clauses. The pattern in each clause corresponds to the right-hand side of a grammar rule. For example, the pattern `(Prim '+ (list e1 e2))` corresponds to the right-hand side `(Prim '+ (exp exp))`. When translating from grammars to patterns, replace non-terminals such as *exp* with pattern variables of your choice (e.g. `e1` and `e2`).

1.4 Recursive Functions

Programs are inherently recursive. For example, an expression is often made of smaller expressions. Thus, the natural way to process an entire program is with a recursive function. As a first example of such a recursive function, we define the function `is_exp` in Figure 1.3, which takes an arbitrary value and determines whether or not it is an expression in \mathcal{L}_{Int} . We say that a function is defined by *structural recursion* when it is defined using a sequence of match clauses that correspond to a grammar, and the body of each clause makes a recursive call on each child node.⁴ Figure 1.3 also defines `is_Lint`, which determines whether an AST is a program in \mathcal{L}_{Int} . In general we can write one recursive function to handle each non-terminal in a grammar. Of the two examples at the bottom of the figure, the first is in \mathcal{L}_{Int} and the second is not.

1.5 Interpreters

The behavior of a program is defined by the specification of the programming language. For example, the Scheme language is defined in the report by Sperber et al. 2009. The Racket language is defined in its reference manual (Flatt and PLT 2014). In this book we use interpreters to specify each language that we

4. This principle of structuring code according to the data definition is advocated in the book *How to Design Programs* by Felleisen et al. (2001).

```

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

(define (is_Lint ast)
  (match ast
    [(Program '() e) (is_exp e)]
    [else #f]))

(is_Lint (Program '() ast1_1))
(is_Lint (Program '()
  (Prim '* (list (Prim 'read '())
    (Prim '+ (list (Int 8)))))))

```

Figure 1.3

Example of recursive functions for \mathcal{L}_{Int} . These functions recognize whether an AST is in \mathcal{L}_{Int} .

consider. An interpreter that is designated as the definition of a language is called a *definitional interpreter* (Reynolds 1972). We warm up by creating a definitional interpreter for the \mathcal{L}_{Int} language. This interpreter serves as a second example of structural recursion. The `interp_Lint` function is defined in Figure 1.4. The body of the function is a match on the input program followed by a call to the `interp_exp` helper function, which in turn has one match clause per grammar rule for \mathcal{L}_{Int} expressions.

Let us consider the result of interpreting a few \mathcal{L}_{Int} programs. The following program adds two integers.

```
(+ 10 32)
```

The result is 42, the answer to life, the universe, and everything: 42!⁵ We wrote the above program in concrete syntax whereas the parsed abstract syntax is:

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

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

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

What is the result of the above program?

As mentioned previously, the \mathcal{L}_{Int} language does not support arbitrarily-large integers, but only 63-bit integers, so we interpret the arithmetic operations of \mathcal{L}_{Int}

5. *The Hitchhiker's Guide to the Galaxy* by Douglas Adams.

```

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

(define (interp_Lint p)
  (match p
    [(Program '() e) (interp_exp e)]))

```

Figure 1.4

Interpreter for the \mathcal{L}_{Int} language.

using fixnum arithmetic in Racket. Suppose

$$n = 9999999999999999999$$

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

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

It produces an error:

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

We establish the convention that if running the definitional interpreter on a program produces an error then the meaning of that program is *unspecified*, unless the error is a **trapped-error**. A compiler for the language is under no obligations regarding programs with unspecified behavior; it does not have to produce an executable, and if it does, that executable can do anything. On the other hand, if the error is a **trapped-error**, then the compiler must produce an executable and it is required to report that an error occurred. To signal an error, exit with a return code of 255. The interpreters in chapters 9 and 10 use **trapped-error**.

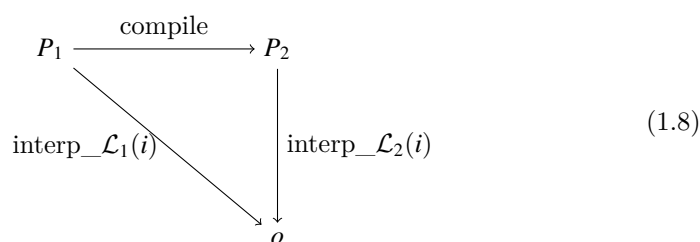
Moving on to the last feature of the \mathcal{L}_{Int} language, the **read** operation prompts the user of the program for an integer. Recall that program (1.1) requests an integer input and then subtracts 8. So if we run

```
(interp_Lint (Program '() ast1_1))
```

and if the input is 50, the result is 42.

We include the `read` operation in \mathcal{L}_{Int} so a clever student cannot implement a compiler for \mathcal{L}_{Int} that simply runs the interpreter during compilation to obtain the output and then generates the trivial code to produce the output.⁶

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 a definitional interpreter for each language. Given a compiler that translates from language \mathcal{L}_1 to \mathcal{L}_2 and given any program P_1 in \mathcal{L}_1 , the compiler must translate it into some program P_2 such that interpreting P_1 and P_2 on their respective interpreters with same input i yields the same output o .



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

1.6 Example Compiler: a Partial Evaluator

In this section we consider a compiler that translates \mathcal{L}_{Int} programs into \mathcal{L}_{Int} programs that may be more efficient. The compiler eagerly computes the parts of the program that do not depend on any inputs, a process known as *partial evaluation* (Jones, Gomard, and Sestoft 1993). For example, given the following program

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

our compiler translates it into the program

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

Figure 1.5 gives the code for a simple partial evaluator for the \mathcal{L}_{Int} language. The output of the partial evaluator is a program in \mathcal{L}_{Int} . In Figure 1.5, the structural recursion over *exp* is captured in the `pe_exp` function whereas the code for partially evaluating the negation and addition operations is factored into three auxiliary functions: `pe_neg`, `pe_add` and `pe_sub`. The input to these functions is the output of partially evaluating the children. The `pe_neg`, `pe_add` and `pe_sub` functions check whether their arguments are integers and if they are, perform the appropriate arithmetic. Otherwise, they create an AST node for the arithmetic operation.

To gain some confidence that the partial evaluator is correct, we can test whether it produces programs that produce the same result as the input programs. That is,

6. Yes, a clever student did this in the first instance of this course!

```

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

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

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

(define (pe_exp e)
  (match e
    [(Int n) (Int n)]
    [(Prim 'read '()) (Prim 'read '())]
    [(Prim '- (list e1)) (pe_neg (pe_exp e1))]
    [(Prim '+ (list e1 e2)) (pe_add (pe_exp e1) (pe_exp e2))]
    [(Prim '- (list e1 e2)) (pe_sub (pe_exp e1) (pe_exp e2))]))

(define (pe_Lint p)
  (match p
    [(Program '() e) (Program '() (pe_exp e))]))

```

Figure 1.5

A partial evaluator for \mathcal{L}_{Int} .

we can test whether it satisfies Diagram 1.8. The following code runs the partial evaluator on several examples and tests the output program. The `parse-program` and `assert` functions are defined in Appendix A.2.

```

(define (test_pe p)
  (assert "testing pe_Lint"
    (equal? (interp_Lint p) (interp_Lint (pe_Lint p)))))

(test_pe (parse-program `(program () (+ 10 (- (+ 5 3))))))
(test_pe (parse-program `(program () (+ 1 (+ 3 1))))))
(test_pe (parse-program `(program () (- (+ 3 (- 5))))))

```

Exercise 1 Create three programs in the \mathcal{L}_{Int} language and test whether partially evaluating them with `pe_Lint` and then interpreting them with `interp_Lint` gives the same result as directly interpreting them with `interp_Lint`.

2 Integers and Variables

This chapter is about compiling a subset of Racket to x86-64 assembly code (Intel 2015). The subset, named \mathcal{L}_{Var} , includes integer arithmetic and local variables. We often refer to x86-64 simply as x86. The chapter begins with a description of the \mathcal{L}_{Var} language (Section 2.1) followed by an introduction to x86 assembly (Section 2.2). The x86 assembly language is large so we discuss only the instructions needed for compiling \mathcal{L}_{Var} . We introduce more x86 instructions in later chapters. After introducing \mathcal{L}_{Var} and x86, we reflect on their differences and come up with a plan to break down the translation from \mathcal{L}_{Var} to x86 into a handful of steps (Section 2.3). The rest of the sections in this chapter give detailed hints regarding each step. We hope to give enough hints that the well-prepared reader, together with a few friends, can implement a compiler from \mathcal{L}_{Var} to x86 in a short time. To give the reader a feeling for the scale of this first compiler, the instructor solution for the \mathcal{L}_{Var} compiler is approximately 500 lines of code.

2.1 The \mathcal{L}_{Var} Language

The \mathcal{L}_{Var} language extends the \mathcal{L}_{Int} language with variables. The concrete syntax of the \mathcal{L}_{Var} language is defined by the grammar in Figure 2.1 and the abstract syntax is defined in Figure 2.2. The non-terminal *var* may be any Racket identifier. As in \mathcal{L}_{Int} , **read** is a nullary operator, **-** is a unary operator, and **+** is a binary operator. Similar to \mathcal{L}_{Int} , the abstract syntax of \mathcal{L}_{Var} includes the **Program** struct to mark the top of the program. Despite the simplicity of the \mathcal{L}_{Var} language, it is rich enough to exhibit several compilation techniques.

Let us dive further into the syntax and semantics of the \mathcal{L}_{Var} language. The **let** feature defines a variable for use within its body and initializes the variable with the value of an expression. The abstract syntax for **let** is defined in Figure 2.2. The concrete syntax for **let** is

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

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

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

$$\begin{array}{lcl}
type & ::= & \text{Integer} \\
exp & ::= & int \mid (\text{read}) \mid (-\ exp) \mid (+\ exp\ exp) \mid (-\ exp\ exp) \\
\hline
exp & ::= & var \mid (\text{let}\ ([var\ exp])\ exp) \\
\mathcal{L}_{\text{Var}} & ::= & exp
\end{array}$$
Figure 2.1The concrete syntax of \mathcal{L}_{Var} .
$$\begin{array}{lcl}
type & ::= & \text{Integer} \\
exp & ::= & (\text{Int}\ int) \mid (\text{Prim}\ 'read'\ ()) \\
& & \mid (\text{Prim}\ '-\ (exp)) \mid (\text{Prim}\ '+\ (exp\ exp)) \mid (\text{Prim}\ '-\ (exp\ exp)) \\
\hline
exp & ::= & (\text{Var}\ var) \mid (\text{Let}\ var\ exp\ exp) \\
\mathcal{L}_{\text{Var}} & ::= & (\text{Program}\ '()\ exp)
\end{array}$$
Figure 2.2The abstract syntax of \mathcal{L}_{Var} .

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 two variables named `x`. Can you figure out the result?

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

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

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

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

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

2.1.1 Extensible Interpreters via Method Overriding

To prepare for discussing the interpreter of \mathcal{L}_{Var} , we explain why we implement it in an object-oriented style. Throughout this book we define many interpreters, one for each of language that we study. Because each language builds on the prior one, there is a lot of commonality between these interpreters. We want to write down the common parts just once instead of many times. A naive interpreter for \mathcal{L}_{Var} would handle the cases for variables and `let` but dispatch to an interpreter for \mathcal{L}_{Int}

in the rest of the cases. The following code sketches this idea. (We explain the `env` parameter soon, in Section 2.1.2.)

```

(define ((interp_Lvar env) e)
  (match e
    [(Var x)
     (dict-ref env x)]
    [(Let x e body)
     (define v ((interp_exp env) e))
     (define env^ (dict-set env x v))
     ((interp_exp env^) body)]
    [else ((interp_Lint env) e)]))

(define ((interp_Lint env) e)
  (match e
    [(Prim '- (list e1))
     (fx- 0 ((interp_Lint env) e1))]
    ...))

```

The problem with this naive approach is that it does not handle situations in which an \mathcal{L}_{Var} feature, such as a variable, is nested inside an \mathcal{L}_{Int} feature, like the `-` operator, as in the following program.

```
(Let 'y (Int 10) (Prim '- (list (Var 'y))))
```

If we invoke `interp_Lvar` on this program, it dispatches to `interp_Lint` to handle the `-` operator, but then it recursively calls `interp_Lint` again on its argument. But there is no case for `Var` in `interp_Lint` so we get an error!

To make our interpreters extensible we need something called *open recursion*, where the tying of the recursive knot is delayed to when the functions are composed. Object-oriented languages provide open recursion via method overriding. The following code uses method overriding to interpret \mathcal{L}_{Int} and \mathcal{L}_{Var} using the `class` feature of Racket. We define one class for each language and define a method for interpreting expressions inside each class. The class for \mathcal{L}_{Var} inherits from the class for \mathcal{L}_{Int} and the method `interp_exp` in \mathcal{L}_{Var} overrides the `interp_exp` in \mathcal{L}_{Int} . Note that the default case of `interp_exp` in \mathcal{L}_{Var} uses `super` to invoke `interp_exp`, and because \mathcal{L}_{Var} inherits from \mathcal{L}_{Int} , that dispatches to the `interp_exp` in \mathcal{L}_{Int} .

```

(define interp-Lvar-class
  (class interp-Lint-class
    (define/override ((interp_exp env) e)
      (match e
        [(Var x)
         (dict-ref env x)]
        [(Let x e body)
         (define v ((interp_exp env) e))
         (define env^ (dict-set env x v))
         ((interp_exp env^) body)]
        [else
         (super (interp_exp env) e)]))
    ...
  ))

(define interp-Lint-class
  (class object%
    (define/public ((interp_exp env) e)
      (match e
        [(Prim '- (list e))
         (fx- 0 ((interp_exp env) e))]
        ...))
    ...))

```

Getting back to the troublesome example, repeated here:

```
(Let 'y (Int 10) (Prim '- (Var 'y)))
```

Association Lists as Dictionaries

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

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

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

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

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

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

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

Figure 2.3

Association lists implement the dictionary interface.

We can invoke the `interp_exp` method for \mathcal{L}_{Var} on this expression, call it `e0`, by creating an object of the \mathcal{L}_{Var} class and calling the `interp_exp` method.

```
((send (new interp-Lvar-class) interp_exp '()) e0)
```

To process the `-` operator, the default case of `interp_exp` in \mathcal{L}_{Var} dispatches to the `interp_exp` method in \mathcal{L}_{Int} . But then for the recursive method call, it dispatches back to `interp_exp` in \mathcal{L}_{Var} , where the `Var` node is handled correctly. Thus, method overriding gives us the open recursion that we need to implement our interpreters in an extensible way.

2.1.2 Definitional Interpreter for \mathcal{L}_{Var}

Having justified the use of classes and methods to implement interpreters, we revisit the definitional interpreter for \mathcal{L}_{Int} in Figure 2.4 and then extend it to create an interpreter for \mathcal{L}_{Var} in Figure 2.5. The interpreter for \mathcal{L}_{Var} adds two new `match` cases for variables and `let`. For `let` we need a way to communicate the value bound to a variable to all the uses of the variable. To accomplish this, we maintain a mapping from variables to values called an *environment*. We use an association list (alist) to represent the environment. Figure 2.3 gives a brief introduction to alists and the `racket/dict` package. The `interp_exp` function takes the current environment, `env`, as an extra parameter. When the interpreter encounters a variable, it looks up the corresponding value in the dictionary. When the interpreter encounters a `Let`, it evaluates the initializing expression, extends the environment with the result value bound to the variable, using `dict-set`, then evaluates the body of the `Let`.

```

(define interp-Lint-class
  (class object%
    (super-new)

    (define/public ((interp_exp env) e)
      (match e
        [(Int n) n]
        [(Prim 'read '())
         (define r (read))
         (cond [(fixnum? r) r]
               [else (error 'interp_exp "expected an integer" r)])])
        [(Prim '- (list e)) (fx- 0 ((interp_exp env) e))]
        [(Prim '+ (list e1 e2))
         (fx+ ((interp_exp env) e1) ((interp_exp env) e2))]
        [(Prim '- (list e1 e2))
         (fx- ((interp_exp env) e1) ((interp_exp env) e2))])

    (define/public (interp_program p)
      (match p
        [(Program '() e) ((interp_exp '()) e)])
      ))

```

Figure 2.4

Interpreter for \mathcal{L}_{Int} as a class.

```

(define interp-Lvar-class
  (class interp-Lint-class
    (super-new)

    (define/override ((interp_exp env) e)
      (match e
        [(Var x) (dict-ref env x)]
        [(Let x e body)
         (define new-env (dict-set env x ((interp_exp env) e)))
         ((interp_exp new-env) body)]
        [else ((super interp_exp env) e)])
      ))

    (define (interp_Lvar p)
      (send (new interp-Lvar-class) interp_program p))

```

Figure 2.5

Interpreter for the \mathcal{L}_{Var} language.

The goal for this chapter is to implement a compiler that translates any program P_1 written in the \mathcal{L}_{Var} language into an x86 assembly program P_2 such that P_2 exhibits the same behavior when run on a computer as the P_1 program interpreted by `interp_Lvar`. That is, they output the same integer n . We depict this correctness

```

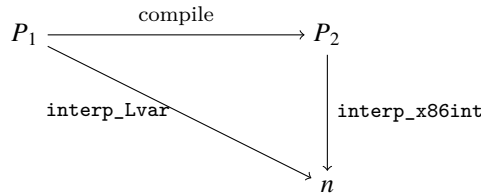
    reg ::= rsp | rbp | rax | rbx | rcx | rdx | rsi | rdi |
           r8 | r9 | r10 | r11 | r12 | r13 | r14 | r15
    arg ::= $int | %reg | int(%reg)
    instr ::= addq arg, arg | subq arg, arg | negq arg | movq arg, arg |
              pushq arg | popq arg | callq label | retq | jmp label |
              label: instr
    x86Int ::= .globl main
                main: instr ...

```

Figure 2.6

The syntax of the x86_{Int} assembly language (AT&T syntax).

criteria in the following diagram.



Next we introduce the x86_{Int} subset of x86 that suffices for compiling \mathcal{L}_{Var} .

2.2 The x86_{Int} Assembly Language

Figure 2.6 defines the concrete syntax for x86_{Int}. We use the AT&T syntax expected by the GNU assembler. A program begins with a `main` label followed by a sequence of instructions. The `globl` directive says that the `main` procedure is externally visible, which is necessary so that the operating system can call it. An x86 program is stored in the computer's memory. For our purposes, the computer's memory is a mapping of 64-bit addresses to 64-bit values. The computer has a *program counter* (PC) stored in the `rip` register that points to the address of the next instruction to be executed. For most instructions, the program counter is incremented after the instruction is executed, so it points to the next instruction in memory. Most x86 instructions take two operands, where each operand is either an integer constant (called an *immediate value*), a *register*, or a memory location.

A register is a special kind of variable that holds a 64-bit value. There are 16 general-purpose registers in the computer and their names are given in Figure 2.6. A register is written with a `%` followed by the register name, such as `%rax`.

An immediate value is written using the notation `$n` where `n` is an integer. An access to memory is specified using the syntax `n(%r)`, which obtains the address stored in register `r` and then adds `n` bytes to the address. The resulting address is used to load or store to memory depending on whether it occurs as a source or destination argument of an instruction.

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

Figure 2.7

An x86 program that computes $(+ 10\ 32)$.

An arithmetic instruction such as `addq s, d` reads from the source s and destination d , applies the arithmetic operation, then writes the result back to the destination d . The move instruction `movq s, d` reads from s and stores the result in d . The `callq $label$` instruction jumps to the procedure specified by the label and `retq` returns from a procedure to its caller. We discuss procedure calls in more detail later in this chapter and in Chapter 7. The last letter `q` indicates that these instructions operate on quadwords, i.e., 64-bit values. The instruction `jmp $label$` updates the program counter to the address of the instruction after the specified label.

Appendix A.3 contains a quick-reference for all of the x86 instructions used in this book.

Figure 2.7 depicts an x86 program that computes $(+ 10\ 32)$. The instruction `movq $10, %rax` puts 10 into register `rax` and then `addq $32, %rax` adds 32 to the 10 in `rax` and puts the result, 42, back into `rax`. The last instruction `retq` finishes the `main` function by returning the integer in `rax` to the operating system. The operating system interprets this integer as the program's exit code. By convention, an exit code of 0 indicates that a program completed successfully, and all other exit codes indicate various errors. Nevertheless, in this book we return the result of the program as the exit code.

We exhibit the use of memory for storing intermediate results in the next example. Figure 2.8 lists an x86 program that computes $(+ 52\ (- 10))$. This program uses a region of memory called the *procedure call stack* (or *stack* for short). The stack consists of a separate *frame* for each procedure call. The memory layout for an individual frame is shown in Figure 2.9. The register `rsp` is called the *stack pointer* and it contains the address of the item at the top of the stack. In general, we use the term *pointer* for something that contains an address. The stack grows downward in memory, so we increase the size of the stack by subtracting from the stack pointer. In the context of a procedure call, the *return address* is the instruction after the call instruction on the caller side. The function call instruction, `callq`, pushes the return address onto the stack prior to jumping to the procedure. The register `rbp` is the *base pointer* and is used to access variables that are stored in the frame of the current procedure call. The base pointer of the caller is stored after the return address. In Figure 2.9 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 2.8, consider how control is transferred from the operating system to the `main` function. The operating system issues a `callq main` instruction which pushes its return address on the stack and then

```

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

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

```

Figure 2.8

An x86 program that computes $(+ 52 (- 10))$.

Position	Contents
8(%rbp)	return address
0(%rbp)	old <code>rbp</code>
-8(%rbp)	variable 1
-16(%rbp)	variable 2
...	...
0(%rsp)	variable n

Figure 2.9

Memory layout of a frame.

jumps to `main`. In x86-64, the stack pointer `rsp` must be divisible by 16 bytes prior to the execution of any `callq` instruction, so when control arrives at `main`, the `rsp` is 8 bytes out of alignment (because the `callq` pushed the return address). The first three instructions are the typical *prelude* for a procedure. The instruction `pushq %rbp` first subtracts 8 from the stack pointer `rsp` and then saves the base pointer of the caller at address `rsp` on the stack. The next instruction `movq %rsp, %rbp` sets the base pointer to the current stack pointer, which is pointing at the location of the old base pointer. The instruction `subq $16, %rsp` moves the stack pointer down to make enough room for storing variables. This program needs one variable (8 bytes) but we round up to 16 bytes so that `rsp` is 16-byte aligned and we're ready to make calls to other functions. The last instruction of the prelude is `jmp start`, which transfers control to the instructions that were generated from the expression $(+ 52 (- 10))$.

The first instruction under the `start` label is `movq $10, -8(%rbp)`, which stores 10 in variable 1. The instruction `negq -8(%rbp)` changes the contents of variable

```

reg    ::=  rsp | rbp | rax | rbx | rcx | rdx | rsi | rdi |
           r8 | r9 | r10 | r11 | r12 | r13 | r14 | r15
arg    ::=  (Imm int) | (Reg reg) | (Deref reg int)
instr  ::=  (Instr addq (arg arg)) | (Instr subq (arg arg)) | (Instr negq (arg))
           | (Instr movq (arg arg)) | (Instr pushq (arg)) | (Instr popq (arg))
           | (Callq label int) | (Retq) | (Jump label)
block  ::=  (Block info (instr ...))
x86Int ::=  (X86Program info ((label . block) ...))

```

Figure 2.10

The abstract syntax of `x86Int` assembly.

1 to -10. The next instruction moves the -10 from variable 1 into the `rax` register. Finally, `addq $52, %rax` adds 52 to the value in `rax`, updating its contents to 42.

The three instructions under the label `conclusion` are the typical *conclusion* of a procedure. The first two restore the `rsp` and `rbp` registers to the state they were in at the beginning of the procedure. In particular, `addq $16, %rsp` moves the stack pointer back to point at the old base pointer. Then `popq %rbp` restores the old base pointer to `rbp` and adds 8 to the stack pointer. The last instruction, `retq`, jumps back to the procedure that called this one and adds 8 to the stack pointer.

Our compiler needs a convenient representation for manipulating x86 programs, so we define an abstract syntax for x86 in Figure 2.10. We refer to this language as `x86Int`. The main difference compared to the concrete syntax of `x86Int` (Figure 2.6) is that labels are not allowed in front of every instruction. Instead instructions are grouped into *basic blocks* with a label associated with every basic block, which is why the `X86Program` struct includes an alist mapping labels to basic blocks. The reason for this organization becomes apparent in Chapter 4 when we introduce conditional branching. The `Block` structure includes an *info* field that is not needed for this chapter but becomes useful in Chapter 3. For now, the *info* field should contain an empty list. Regarding the abstract syntax for `callq`, the `Callq` AST node includes an integer for representing the arity of the function, i.e., the number of arguments, which is helpful to know during register allocation (Chapter 3).

2.3 Planning the trip to x86

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

1. x86 arithmetic instructions typically have two arguments and update the second argument in place. In contrast, \mathcal{L}_{Var} arithmetic operations take two arguments

and produce a new value. An x86 instruction may have at most one memory-accessing argument. Furthermore, some x86 instructions place special restrictions on their arguments.

2. An argument of an \mathcal{L}_{Var} operator can be a deeply-nested expression, whereas x86 instructions restrict their arguments to be integer constants, registers, and memory locations.
3. The order of execution in x86 is explicit in the syntax: a sequence of instructions and jumps to labeled positions, whereas in \mathcal{L}_{Var} the order of evaluation is a left-to-right depth-first traversal of the abstract syntax tree.
4. A program in \mathcal{L}_{Var} can have any number of variables whereas x86 has 16 registers and the procedure call stack.
5. Variables in \mathcal{L}_{Var} can shadow other variables with the same name. In x86, registers have unique names and memory locations have unique addresses.

We ease the challenge of compiling from \mathcal{L}_{Var} to x86 by breaking down the problem into several steps, dealing with the above differences one at a time. Each of these steps is called a *pass* of the compiler. This terminology comes from the way each step passes over, or traverses, the AST of the program. Furthermore, we follow the nanopass approach, which means we strive for each pass to accomplish one clear objective (not two or three at the same time). We begin by sketching how we might implement each pass, and give them names. We then figure out an ordering of the passes and the input/output language for each pass. The very first pass has \mathcal{L}_{Var} as its input language and the last pass has x86_{Int} as its output language. In between we can choose whichever language is most convenient for expressing the output of each pass, whether that be \mathcal{L}_{Var} , x86_{Int} , or new *intermediate languages* of our own design. Finally, to implement each pass we write one recursive function per non-terminal in the grammar of the input language of the pass.

Our compiler for \mathcal{L}_{Var} consists of the following passes.

uniquify deals with the shadowing of variables by renaming every variable to a unique name.

remove_complex_operands ensures that each subexpression of a primitive operation or function call is a variable or integer, that is, an *atomic* expression. We refer to non-atomic expressions as *complex*. This pass introduces temporary variables to hold the results of complex subexpressions.

explicate_control makes the execution order of the program explicit. It converts the abstract syntax tree representation into a graph in which each node is a labeled sequence of statements and the edges are **goto** statements.

select_instructions handles the difference between \mathcal{L}_{Var} operations and x86 instructions. This pass converts each \mathcal{L}_{Var} operation to a short sequence of instructions that accomplishes the same task.

assign_homes replaces variables with registers or stack locations.

Our treatment of `remove_complex_operands` and `explicate_control` as separate passes is an example of the nanopass approach⁷. The traditional approach is to combine them into a single step (Aho et al. 2006).

The next question is: in what order should we apply these passes? This question can be challenging because it is difficult to know ahead of time which orderings will be better (easier to implement, produce more efficient code, etc.) so oftentimes trial-and-error is involved. Nevertheless, we can plan ahead and make educated choices regarding the ordering.

What should be the ordering of `explicate_control` with respect to `uniquify`? The `uniquify` pass should come first because `explicate_control` changes all the `let`-bound variables to become local variables whose scope is the entire program, which would confuse variables with the same name. We place `remove_complex_operands` before `explicate_control` because the later removes the `let` form, but it is convenient to use `let` in the output of `remove_complex_operands`. The ordering of `uniquify` with respect to `remove_complex_operands` does not matter so we arbitrarily choose `uniquify` to come first.

The `select_instructions` and `assign_homes` passes are intertwined. In Chapter 7 we learn that, in x86, registers are used for passing arguments to functions and it is preferable to assign parameters to their corresponding registers. This suggests that it would be better to start with the `select_instructions` pass, which generates the instructions for argument passing, before performing register allocation. On the other hand, by selecting instructions first we may run into a dead end in `assign_homes`. Recall that only one argument of an x86 instruction may be a memory access but `assign_homes` might be forced to assign both arguments to memory locations. A sophisticated approach is to repeat the two passes until a solution is found. However, to reduce implementation complexity we recommend placing `select_instructions` first, followed by the `assign_homes`, then a third pass named `patch_instructions` that uses a reserved register to fix outstanding problems.

Figure 2.11 presents the ordering of the compiler passes and identifies the input and output language of each pass. The output of the `select_instructions` pass is the `x86Var` language, which extends `x86Int` with an unbounded number of program-scope variables and removes the restrictions regarding instruction arguments. The last pass, `prelude_and_conclusion`, places the program instructions inside a `main` function with instructions for the prelude and conclusion. In the next section we discuss the `CVar` intermediate language that serves as the output of `explicate_control`. The remainder of this chapter provides guidance on the implementation of each of the compiler passes in Figure 2.11.

7. For analogous decompositions of the translation into continuation passing style, see the work of Lawall and Danvy (1993) and Hatcliff and Danvy (1994).

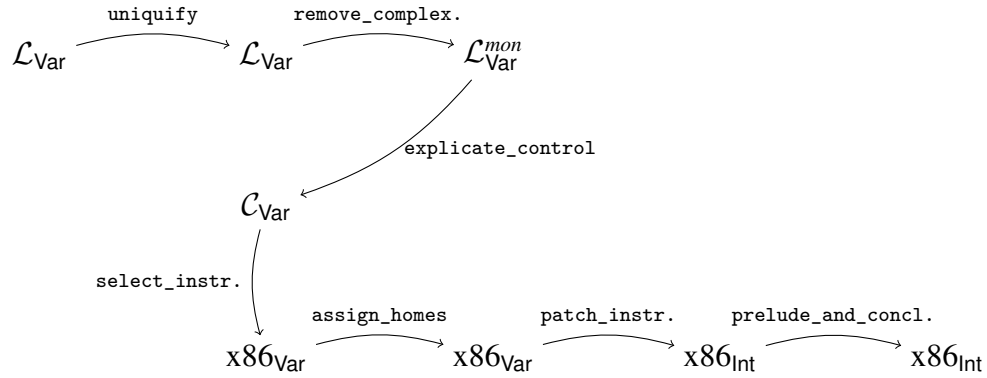


Figure 2.11

Diagram of the passes for compiling \mathcal{L}_{Var} .

2.3.1 The \mathcal{C}_{Var} Intermediate Language

The output of `explicate_control` is similar to the \mathcal{C} language (Kernighan and Ritchie 1988) in that it has separate syntactic categories for expressions and statements, so we name it \mathcal{C}_{Var} . This style of intermediate language is also known as *three-address code*, to emphasize that the typical form of a statement such as $x = (+ y z)$; involves three addresses: x , y , and z (Aho et al. 2006).

The concrete syntax for \mathcal{C}_{Var} is defined in Figure 2.12 and the abstract syntax for \mathcal{C}_{Var} is defined in Figure 2.13. The \mathcal{C}_{Var} language supports the same operators as \mathcal{L}_{Var} but the arguments of operators are restricted to atomic expressions. Instead of `let` expressions, \mathcal{C}_{Var} has assignment statements which can be executed in sequence using the `Seq` form. A sequence of statements always ends with `Return`, a guarantee that is baked into the grammar rules for *tail*. The naming of this non-terminal comes from the term *tail position*, which refers to an expression that is the last one to execute within a function or program.

A \mathcal{C}_{Var} program consists of an alist mapping labels to tails. This is more general than necessary for the present chapter, as we do not yet introduce `goto` for jumping to labels, but it saves us from having to change the syntax in Chapter 4. For now there will be just one label, `start`, and the whole program is its tail. The *info* field of the `CProgram` form, after the `explicate_control` pass, contains a mapping from the symbol `locals` to a list of variables, that is, a list of all the variables used in the program. At the start of the program, these variables are uninitialized; they become initialized on their first assignment.

The definitional interpreter for \mathcal{C}_{Var} is in the support code, in the file `interp-Cvar.rkt`.

2.4 Uniquify Variables

The `uniquify` pass compiles \mathcal{L}_{Var} programs into \mathcal{L}_{Var} programs in which every `let` binds a unique variable name. For example, the `uniquify` pass should translate the program on the left into the program on the right.

```

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

```

Figure 2.12

The concrete syntax of the C_{Var} intermediate language.

```

atm  ::= (Int int) | (Var var)
exp  ::= atm | (Prim 'read ()) | (Prim '- (atm))
      | (Prim '+ (atm atm)) | (Prim '- (atm atm))
stmt ::= (Assign (Var var) exp)
tail  ::= (Return exp) | (Seq stmt tail)
CVar ::= (CProgram info ((label . tail) ...))

```

Figure 2.13

The abstract syntax of the C_{Var} intermediate language.

$$\begin{array}{ccc}
 (\text{let } ([x \ 32]) & \Rightarrow & (\text{let } ([x.1 \ 32]) \\
 (+ (\text{let } ([x \ 10]) x) x)) & & (+ (\text{let } ([x.2 \ 10]) x.2) x.1))
 \end{array}$$

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

$$\begin{array}{ccc}
 (\text{let } ([x (\text{let } ([x \ 4]) & \Rightarrow & (\text{let } ([x.2 (\text{let } ([x.1 \ 4]) \\
 (+ x \ 1))]) & & (+ x.1 \ 1))]) \\
 (+ x \ 2)) & & (+ x.2 \ 2))
 \end{array}$$

We recommend implementing `uniquify` by creating a structurally recursive function named `uniquify_exp` that mostly just copies an expression. However, when encountering a `let`, it should generate a unique name for the variable and associate the old name with the new name in an alist.⁸ The `uniquify_exp` function needs to access this alist when it gets to a variable reference, so we add a parameter to `uniquify_exp` for the alist.

The skeleton of the `uniquify_exp` function is shown in Figure 2.14. The `for/list` form of Racket is useful for transforming the element of a list to produce a new list.

Exercise 2 Complete the `uniquify` pass by filling in the blanks in Figure 2.14, that is, implement the cases for variables and for the `let` form in the file `compiler.rkt` in the support code.

8. The Racket function `gensym` is handy for generating unique variable names.

```

(define (uniquify_exp env)
  (lambda (e)
    (match e
      [(Var x) ___]
      [(Int n) (Int n)]
      [(Let x e body) ___]
      [(Prim op es)
       (Prim op (for/list ([e es]) ((uniquify_exp env) e))))]))

(define (uniquify p)
  (match p
    [(Program '() e) (Program '() ((uniquify_exp '()) e))]))

```

Figure 2.14

Skeleton for the `uniquify` pass.

Exercise 3 Create five \mathcal{L}_{Var} programs that exercise the most interesting parts of the `uniquify` pass, that is, the programs should include `let` forms, variables, and variables that shadow each other. The five programs should be placed in the subdirectory named `tests` and the file names should start with `var_test_` followed by a unique integer and end with the file extension `.rkt`. The `run-tests.rkt` script in the support code checks whether the output programs produce the same result as the input programs. The script uses the `interp-tests` function (Appendix A.2) from `utilities.rkt` to test your `uniquify` pass on the example programs. The `passes` parameter of `interp-tests` is a list that should have one entry for each pass in your compiler. For now, define `passes` to contain just one entry for `uniquify` as shown below.

```

(define passes
  (list (list "uniquify" uniquify interp_Lvar type-check-Lvar)))

```

Run the `run-tests.rkt` script in the support code to check whether the output programs produce the same result as the input programs.

2.5 Remove Complex Operands

The `remove_complex_operands` pass compiles \mathcal{L}_{Var} programs into a restricted form in which the arguments of operations are atomic expressions. Put another way, this pass removes complex operands, such as the expression `(- 10)` in the program below. This is accomplished by introducing a new temporary variable, assigning the complex operand to the new variable, and then using the new variable in place of the complex operand, as shown in the output of `remove_complex_operands` on the right.

<pre>(let ([x (+ 42 (- 10))]) (+ x 10))</pre>	\Rightarrow	<pre>(let ([x (let ([tmp.1 (- 10)]) (+ 42 tmp.1))]) (+ x 10))</pre>
---	---------------	---

```

 $atm ::= (Int\ int) \mid (Var\ var)$ 
 $exp ::= atm \mid (Prim\ 'read\ ())$ 
        $\mid (Prim\ '-\ (atm)) \mid (Prim\ '+\ (atm\ atm)) \mid (Prim\ '-\ (atm\ atm))$ 
        $\mid (Let\ var\ exp\ exp)$ 
 $\mathcal{L}_{Var}^{mon} ::= (Program\ '()\ exp)$ 

```

Figure 2.15

\mathcal{L}_{Var}^{mon} is \mathcal{L}_{Var} with operands restricted to atomic expressions.

Figure 2.15 presents the grammar for the output of this pass, the language \mathcal{L}_{Var}^{mon} . The only difference is that operator arguments are restricted to be atomic expressions that are defined by the *atm* non-terminal. In particular, integer constants and variables are atomic.

The atomic expressions are pure (they do not cause or depend on side-effects) whereas complex expressions may have side effects, such as `(Prim 'read ())`. A language with this separation between pure versus side-effecting expressions is said to be in monadic normal form (Moggi 1991; Danvy 2003) which explains the *mon* in the name \mathcal{L}_{Var}^{mon} . An important invariant of the `remove_complex_operands` pass is that the relative ordering among complex expressions is not changed, but the relative ordering between atomic expressions and complex expressions can change and often does. The reason that these changes are behavior preserving is that the atomic expressions are pure.

Another well-known form for intermediate languages is the *administrative normal form* (ANF) (Danvy 1991; Flanagan et al. 1993). The \mathcal{L}_{Var}^{mon} language is not quite in ANF because we allow the right-hand side of a `let` to be a complex expression.

We recommend implementing this pass with two mutually recursive functions, `rco_atom` and `rco_exp`. The idea is to apply `rco_atom` to subexpressions that need to become atomic and to apply `rco_exp` to subexpressions that do not. Both functions take an \mathcal{L}_{Var} expression as input. The `rco_exp` function returns an expression. The `rco_atom` function returns two things: an atomic expression and an alist mapping temporary variables to complex subexpressions. You can return multiple things from a function using Racket's `values` form and you can receive multiple things from a function call using the `define-values` form.

Returning to the example program with the expression `(+ 42 (- 10))`, the subexpression `(- 10)` should be processed using the `rco_atom` function because it is an argument of the `+` operator and therefore needs to become atomic. The output of `rco_atom` applied to `(- 10)` is as follows.

$$(-\ 10) \quad \Rightarrow \quad \begin{array}{l} tmp.1 \\ ((tmp.1\ .\ (-\ 10))) \end{array}$$

Take special care of programs such as the following that bind a variable to an atomic expression. You should leave such variable bindings unchanged, as shown

in the program on the right

```

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

```

A careless implementation might produce the following output with unnecessary temporary variables.

```

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

```

Exercise 4 Implement the `remove_complex_operands` function in `compiler.rkt`. Create three new \mathcal{L}_{Var} programs that exercise the interesting code in the `remove_complex_operands` pass. Follow the guidelines regarding file names described in Exercise 3. In the `run-tests.rkt` script, add the following entry to the list of `passes` and then run the script to test your compiler.

```
(list "remove-complex" remove_complex_operands interp_Lvar type-check-Lvar)
```

While debugging your compiler, it is often useful to see the intermediate programs that are output from each pass. To print the intermediate programs, place `(debug-level 1)` before the call to `interp-tests` in `run-tests.rkt`.

2.6 Explicate Control

The `explicate_control` pass compiles \mathcal{L}_{Var} programs into \mathcal{C}_{Var} programs that make the order of execution explicit in their syntax. For now this amounts to flattening `let` constructs into a sequence of assignment statements. For example, consider the following \mathcal{L}_{Var} program.

```

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

```

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

```

(define (explicate_tail e)
  (match e
    [(Var x) ___]
    [(Int n) (Return (Int n))]
    [(Let x rhs body) ___]
    [(Prim op es) ___]
    [else (error "explicate_tail unhandled case" e)]))

(define (explicate_assign e x cont)
  (match e
    [(Var x) ___]
    [(Int n) (Seq (Assign (Var x) (Int n)) cont)]
    [(Let y rhs body) ___]
    [(Prim op es) ___]
    [else (error "explicate_assign unhandled case" e)]))

(define (explicate_control p)
  (match p
    [(Program info body) ___]))

```

Figure 2.16

Skeleton for the `explicate_control` pass.

<pre> (let ([y (let ([x.1 20]) (let ([x.2 22]) (+ x.1 x.2)))]]) y) </pre>	\Rightarrow	<pre> start: x.1 = 20; x.2 = 22; y = (+ x.1 x.2); return y; </pre>
---	---------------	--

The organization of this pass depends on the notion of tail position that we have alluded to earlier. Here is the definition.

Definition 1 *The following rules define when an expression is in **tail position** for the language \mathcal{L}_{Var} .*

1. In $(\text{Program } () e)$, expression e is in tail position.
2. If $(\text{Let } x e_1 e_2)$ is in tail position, then so is e_2 .

We recommend implementing `explicate_control` using two recursive functions, `explicate_tail` and `explicate_assign`, as suggested in the skeleton code in Figure 2.16. The `explicate_tail` function should be applied to expressions in tail position whereas the `explicate_assign` should be applied to expressions that occur on the right-hand-side of a `let`. The `explicate_tail` function takes an *exp* in \mathcal{L}_{Var} as input and produces a *tail* in \mathcal{C}_{Var} (see Figure 2.13). The `explicate_assign` function takes an *exp* in \mathcal{L}_{Var} , the variable that it is to be assigned to, and a *tail* in \mathcal{C}_{Var} for the code that comes after the assignment. The `explicate_assign` function returns a *tail* in \mathcal{C}_{Var} .

The `explicate_assign` function is in accumulator-passing style: the `cont` parameter is used for accumulating the output. This accumulator-passing style plays an

important role in how we generate high-quality code for conditional expressions in Chapter 4. The abbreviation `cont` is for continuation because it contains the generated code that should come after the current assignment. This code organization is also related to continuation-passing style, except that `cont` is not what happens next during compilation, but what happens next in the generated code.

Exercise 5 Implement the `explicate_control` function in `compiler.rkt`. Create three new \mathcal{L}_{Int} programs that exercise the code in `explicate_control`. In the `run-tests.rkt` script, add the following entry to the list of `passes` and then run the script to test your compiler.

```
(list "explicate control" explicate_control interp_Cvar type-check-Cvar)
```

2.7 Select Instructions

In the `select_instructions` pass we begin the work of translating from C_{Var} to $x86_{\text{Var}}$. The target language of this pass is a variant of x86 that still uses variables, so we add an AST node of the form $(\text{Var } var)$ to the *arg* non-terminal of the $x86_{\text{Int}}$ abstract syntax (Figure 2.10). We recommend implementing the `select_instructions` with three auxiliary functions, one for each of the non-terminals of C_{Var} : *atm*, *stmt*, and *tail*.

The cases for *atm* are straightforward; variables stay the same and integer constants change to immediates, that is, $(\text{Int } n)$ changes to $(\text{Imm } n)$.

Next consider the cases for the *stmt* non-terminal, starting with arithmetic operations. For example, consider the addition operation below, on the left side. There is an `addq` instruction in x86, but it performs an in-place update. So we could move *arg*₁ into the left-hand side *var* and then add *arg*₂ to *var*, where *arg*₁ and *arg*₂ are the translations of *atm*₁ and *atm*₂ respectively.

$$var = (+ \text{atm}_1 \text{atm}_2); \quad \Rightarrow \quad \begin{array}{l} \text{movq } arg_1, var \\ \text{addq } arg_2, var \end{array}$$

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

$$var = (+ \text{atm}_1 var); \quad \Rightarrow \quad \text{addq } arg_1, var$$

The `read` operation does not have a direct counterpart in x86 assembly, so we provide this functionality with the function `read_int` in the file `runtime.c`, written in C (Kernighan and Ritchie 1988). In general, we refer to all of the functionality in this file as the *runtime system*, or simply the *runtime* for short. When compiling your generated x86 assembly code, you need to compile `runtime.c` to `runtime.o` (an “object file”, using `gcc` with option `-c`) and link it into the executable. For

our purposes of code generation, all you need to do is translate an assignment of `read` into a call to the `read_int` function followed by a move from `rax` to the left-hand-side variable. (Recall that the return value of a function goes into `rax`.)

```
var = (read);           ⇒  callq read_int
                           movq %rax, var
```

There are two cases for the *tail* non-terminal: **Return** and **Seq**. Regarding **Return**, we recommend treating it as an assignment to the `rax` register followed by a jump to the conclusion of the program (so the conclusion needs to be labeled). For (**Seq** *s t*), you can translate the statement *s* and tail *t* recursively and then append the resulting instructions.

Exercise 6 Implement the `select_instructions` pass in `compiler.rkt`. Create three new example programs that are designed to exercise all of the interesting cases in this pass. In the `run-tests.rkt` script, add the following entry to the list of `passes` and then run the script to test your compiler.

```
(list "instruction selection" select_instructions interp_pseudo-x86-0)
```

2.8 Assign Homes

The `assign_homes` pass compiles `x86var` programs to `x86var` programs that no longer use program variables. Thus, the `assign_homes` pass is responsible for placing all of the program variables in registers or on the stack. For runtime efficiency, it is better to place variables in registers, but as there are only 16 registers, some programs must necessarily resort to placing some variables on the stack. In this chapter we focus on the mechanics of placing variables on the stack. We study an algorithm for placing variables in registers in Chapter 3.

Consider again the following \mathcal{L}_{var} program from Section 2.5.

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

The output of `select_instructions` is shown below, on the left, and the output of `assign_homes` is on the right. In this example, we assign variable `a` to stack location `-8(%rbp)` and variable `b` to location `-16(%rbp)`.

```
movq $42, a           ⇒  movq $42, -8(%rbp)
movq a, b              movq -8(%rbp), -16(%rbp)
movq b, %rax           movq -16(%rbp), %rax
```

The `assign_homes` pass should replace all variables with stack locations. The list of variables can be obtained from the `locals-types` entry in the *info* of the `X86Program` node. The `locals-types` entry is an alist mapping all the variables in the program to their types (for now just `Integer`). As an aside, the `locals-types` entry is computed by `type-check-Cvar` in the support code, which installs it in

the *info* field of the `CProgram` node, which you should propagate to the `X86Program` node. In the process of assigning variables to stack locations, it is convenient for you to compute and store the size of the frame (in bytes) in the *info* field of the `X86Program` node, with the key `stack-space`, which is needed later to generate the conclusion of the `main` procedure. The x86-64 standard requires the frame size to be a multiple of 16 bytes.

Exercise 7 Implement the `assign_homes` pass in `compiler.rkt`, defining auxiliary functions for each of the non-terminals in the `x86var` grammar. We recommend that the auxiliary functions take an extra parameter that maps variable names to homes (stack locations for now). In the `run-tests.rkt` script, add the following entry to the list of `passes` and then run the script to test your compiler.

```
(list "assign homes" assign-homes interp_x86-0)
```

2.9 Patch Instructions

The `patch_instructions` pass compiles from `x86var` to `x86int` by making sure that each instruction adheres to the restriction that at most one argument of an instruction may be a memory reference.

We return to the following example.

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

The `assign_homes` pass produces the following translation.

```
movq $42, -8(%rbp)
movq -8(%rbp), -16(%rbp)
movq -16(%rbp), %rax
```

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

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

Exercise 8 Implement the `patch_instructions` pass in `compiler.rkt`. Create three new example programs that are designed to exercise all of the interesting cases in this pass. In the `run-tests.rkt` script, add the following entry to the list of `passes` and then run the script to test your compiler.

```
(list "patch instructions" patch_instructions interp_x86-0)
```

2.10 Generate Prelude and Conclusion

The last step of the compiler from \mathcal{L}_{Var} to x86 is to generate the `main` function with a prelude and conclusion wrapped around the rest of the program, as shown in Figure 2.8 and discussed in Section 2.2.

When running on Mac OS X, your compiler should prefix an underscore to all labels, e.g., changing `main` to `_main`. The Racket call `(system-type 'os)` is useful for determining which operating system the compiler is running on. It returns `'macosx`, `'unix`, or `'windows`.

Exercise 9 Implement the `prelude_and_conclusion` pass in `compiler.rkt`. In the `run-tests.rkt` script, add the following entry to the list of `passes` and then run the script to test your compiler.

```
(list "prelude and conclusion" prelude-and-conclusion interp_x86-0)
```

Uncomment the call to the `compiler-tests` function (Appendix A.2), which tests your complete compiler by executing the generated x86 code. It translates the x86 AST that you produce into a string by invoking the `print-x86` method of the `print-x86-class` in `utilities.rkt`. Compile the provided `runtime.c` file to `runtime.o` using `gcc`. Run the script to test your compiler.

2.11 Challenge: Partial Evaluator for \mathcal{L}_{Var}

This section describes two optional challenge exercises that involve adapting and improving the partial evaluator for \mathcal{L}_{Int} that was introduced in Section 1.6.

Exercise 10 Adapt the partial evaluator from Section 1.6 (Figure 1.5) so that it applies to \mathcal{L}_{Var} programs instead of \mathcal{L}_{Int} programs. Recall that \mathcal{L}_{Var} adds variables and `let` binding to the \mathcal{L}_{Int} language, so you will need to add cases for them in the `pe_exp` function. Once complete, add the partial evaluation pass to the front of your compiler and make sure that your compiler still passes all of the tests.

Exercise 11 Improve on the partial evaluator by replacing the `pe_neg` and `pe_add` auxiliary functions with functions that know more about arithmetic. For example, your partial evaluator should translate

(+ 1 (+ (read) 1)) into (+ 2 (read))

To accomplish this, the `pe_exp` function should produce output in the form of the *residual* non-terminal of the following grammar. The idea is that when processing an addition expression, we can always produce either 1) an integer constant, 2) an addition expression with an integer constant on the left-hand side but not the right-hand side, or 3) or an addition expression in which neither subexpression is a constant.

```
inert    ::= var | (read) | (- var) | (- (read)) | (+ inert inert)
          | (let ([var residual]) residual)
residual ::= int | (+ int inert) | inert
```

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

3 Register Allocation

In Chapter 2 we compiled \mathcal{L}_{Var} to x86, storing variables on the procedure call stack. It can take 10s to 100s of cycles for the CPU to access locations on the stack whereas accessing a register takes only a single cycle. In this chapter we improve the efficiency of our generated code by storing some variables in registers. The goal of register allocation is to fit as many variables into registers as possible. Some programs have more variables than registers so we cannot always map each variable to a different register. Fortunately, it is common for different variables to be in-use during different periods of time during program execution, and in those cases we can map multiple variables to the same register.

The program in Figure 3.1 serves as a running example. The source program is on the left and the output of instruction selection is on the right. The program is almost in the x86 assembly language but it still uses variables. Consider variables x and z . After the variable x is moved to z it is no longer in-use. Variable z , on the other hand, is used only after this point, so x and z could share the same register.

Example \mathcal{L}_{Var} program:	After instruction selection:
<pre>(let ([v 1]) (let ([w 42]) (let ([x (+ v 7)]) (let ([y x]) (let ([z (+ x w)]) (+ z (- y))))))</pre>	<pre>locals-types: x : Integer, y : Integer, z : Integer, t : Integer, v : Integer, w : Integer start: movq \$1, v movq \$42, w movq v, x addq \$7, x movq x, y movq x, z addq w, z movq y, t negq t movq z, %rax addq t, %rax jmp conclusion</pre>

Figure 3.1
A running example for register allocation.

The topic of Section 3.2 is how to compute where a variable is in-use. Once we have that information, we compute which variables are in-use at the same time, i.e., which ones *interfere* with each other, and represent this relation as an undirected graph whose vertices are variables and edges indicate when two variables interfere (Section 3.3). We then model register allocation as a graph coloring problem (Section 3.4).

If we run out of registers despite these efforts, we place the remaining variables on the stack, similar to what we did in Chapter 2. It is common to use the verb *spill* for assigning a variable to a stack location. The decision to spill a variable is handled as part of the graph coloring process.

We make the simplifying assumption that each variable is assigned to one location (a register or stack address). A more sophisticated approach is to assign a variable to one or more locations in different regions of the program. For example, if a variable is used many times in short sequence and then only used again after many other instructions, it could be more efficient to assign the variable to a register during the initial sequence and then move it to the stack for the rest of its lifetime. We refer the interested reader to Cooper and Torczon (2011) (Chapter 13) for more information about that approach.

3.1 Registers and Calling Conventions

As we perform register allocation, we must be aware of the *calling conventions* that govern how functions calls are performed in x86. Even though \mathcal{L}_{Var} does not include programmer-defined functions, our generated code includes a `main` function that is called by the operating system and our generated code contains calls to the `read_int` function.

Function calls require coordination between two pieces of code that may be written by different programmers or generated by different compilers. Here we follow the System V calling conventions that are used by the GNU C compiler on Linux and MacOS (Bryant and O'Hallaron 2005; Matz et al. 2013). The calling conventions include rules about how functions share the use of registers. In particular, the caller is responsible for freeing up some registers prior to the function call for use by the callee. These are called the *caller-saved registers* and they are

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

On the other hand, the callee is responsible for preserving the values of the *callee-saved registers*, which are

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

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

- The caller should assume that all the caller-saved registers get overwritten with arbitrary values by the callee. On the other hand, the caller can safely assume that all the callee-saved registers retain their original values.

- The callee can freely use any of the caller-saved registers. However, if the callee wants to use a callee-saved register, the callee must arrange to put the original value back in the register prior to returning to the caller. This can be accomplished by saving the value to the stack in the prelude of the function and restoring the value in the conclusion of the function.

In x86, registers are also used for passing arguments to a function and for the return value. In particular, the first six arguments of a function are passed in the following six registers, in this order.

`rdi rsi rdx rcx r8 r9`

If there are more than six arguments, then the convention is to use space on the frame of the caller for the rest of the arguments. However, in Chapter 7 we arrange never to need more than six arguments. For now, the only function we care about is `read_int` and it takes zero arguments. The register `rax` is used for the return value of a function.

The next question is how these calling conventions impact register allocation. Consider the \mathcal{L}_{Var} program in Figure 3.2. We first analyze this example from the caller point of view and then from the callee point of view. We refer to a variable that is in-use during a function call as being a *call-live variable*.

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

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

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

Returning to the example in Figure 3.2, let us analyze the generated x86 code on the right-hand side. Notice that variable `x` is assigned to `rbx`, a callee-saved register. Thus, it is already in a safe place during the second call to `read_int`. Next, notice

Generated x86 assembly:

```

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

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

```

Example \mathcal{L}_{Var} program:

```

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

```

Figure 3.2

An example with function calls.

that variable *y* is assigned to *rcx*, a caller-saved register, because *y* is not a call-live variable.

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

3.2 Liveness Analysis

The `uncover_live` pass performs *liveness analysis*, that is, it discovers which variables are in-use in different regions of a program. A variable or register is *live* at a program point if its current value is used at some later point in the program. We refer to variables, stack locations, and registers collectively as *locations*. Consider the following code fragment in which there are two writes to *b*. Are variables *a* and *b* both live at the same time?

The Racket Set Package

A *set* is an unordered collection of elements without duplicates. Here are some of the operations defined on sets.

(**set** *v* ...) constructs a set containing the specified elements.

(**set-union** *set*₁ *set*₂) returns the union of the two sets.

(**set-subtract** *set*₁ *set*₂) returns the set difference of the two sets.

(**set-member?** *set* *v*) answers whether element *v* is in *set*.

(**set-count** *set*) returns the number of unique elements in *set*.

(**set->list** *set*) converts *set* to a list.

Figure 3.3

The *set* data structure.

```

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

```

The answer is no because *a* is live from line 1 to 3 and *b* is live from line 4 to 5. The integer written to *b* on line 2 is never used because it is overwritten (line 4) before the next read (line 5).

The live locations for each instruction can be computed by traversing the instruction sequence back to front (i.e., backwards in execution order). Let I_1, \dots, I_n be the instruction sequence. We write $L_{\text{after}}(k)$ for the set of live locations after instruction I_k and $L_{\text{before}}(k)$ for the set of live locations before instruction I_k . We recommend representing these sets with the Racket **set** data structure described in Figure 3.3.

The live locations after an instruction are always the same as the live locations before the next instruction.

$$L_{\text{after}}(k) = L_{\text{before}}(k + 1) \quad (3.9)$$

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

$$L_{\text{after}}(n) = \emptyset \quad (3.10)$$

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

$$L_{\text{before}}(k) = (L_{\text{after}}(k) - W(k)) \cup R(k), \quad (3.11)$$

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

There is a special case for **jmp** instructions. The locations that are live before a **jmp** should be the locations in L_{before} at the target of the jump. So we recommend maintaining an alist named **label->live** that maps each label to the L_{before} for the first instruction in its block. For now the only **jmp** in a *x86_{var}* program is the

	$L_{\text{before}}(1) = \emptyset, L_{\text{after}}(1) = \{a\}$
1 <code>movq \$5, a</code>	$L_{\text{before}}(2) = \{a\}, L_{\text{after}}(2) = \{a\}$
2 <code>movq \$30, b</code>	$L_{\text{before}}(3) = \{a\}, L_{\text{after}}(2) = \{c\}$
3 <code>movq a, c</code>	$L_{\text{before}}(4) = \{c\}, L_{\text{after}}(4) = \{b, c\}$
4 <code>movq \$10, b</code>	$L_{\text{before}}(5) = \{b, c\}, L_{\text{after}}(5) = \emptyset$
5 <code>addq b, c</code>	

Figure 3.4

Example output of liveness analysis on a short example.

jump to the conclusion. (For example, see Figure 3.1.) The conclusion reads from `rax` and `rsp`, so the alist should map **conclusion** to the set `{rax, rsp}`.

Let us walk through the above example, applying these formulas starting with the instruction on line 5. We collect the answers in Figure 3.4. The L_{after} for the `addq b, c` instruction is \emptyset because it is the last instruction (formula 3.10). The L_{before} for this instruction is $\{b, c\}$ because it reads from variables `b` and `c` (formula 3.11), that is

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

Moving on the the instruction `movq $10, b` at line 4, we copy the live-before set from line 5 to be the live-after set for this instruction (formula 3.9).

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

This move instruction writes to `b` and does not read from any variables, so we have the following live-before set (formula 3.11).

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

The live-before for instruction `movq a, c` is $\{a\}$ because it writes to $\{c\}$ and reads from $\{a\}$ (formula 3.11). The live-before for `movq $30, b` is $\{a\}$ because it writes to a variable that is not live and does not read from a variable. Finally, the live-before for `movq $5, a` is \emptyset because it writes to variable `a`.

Exercise 12 Perform liveness analysis by hand on the running example in Figure 3.1, computing the live-before and live-after sets for each instruction. Compare your answers to the solution shown in Figure 3.5.

Exercise 13 Implement the `uncover_live` pass. Store the sequence of live-after sets in the `info` field of the `Block` structure. We recommend creating an auxiliary function that takes a list of instructions and an initial live-after set (typically empty) and returns the list of live-after sets. We recommend creating auxiliary functions to 1) compute the set of locations that appear in an `arg`, 2) compute the locations read by an instruction (the R function), and 3) the locations written by an instruction (the W function). The `callq` instruction should include all of the caller-saved registers in its write-set W because the calling convention says that those registers

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

Figure 3.5

The running example annotated with live-after sets.

may be written to during the function call. Likewise, the `callq` instruction should include the appropriate argument-passing registers in its read-set R , depending on the arity of the function being called. (This is why the abstract syntax for `callq` includes the arity.)

3.3 Build the Interference Graph

Based on the liveness analysis, we know where each location is live. However, during register allocation, we need to answer questions of the specific form: are locations u and v live at the same time? (And therefore cannot be assigned to the same register.) To make this question more efficient to answer, we create an explicit data structure, an *interference graph*. An interference graph is an undirected graph that has an edge between two locations if they are live at the same time, that is, if they interfere with each other. We recommend using the Racket `graph` package (Figure 3.6) to represent the interference graph.

A straightforward way to compute the interference graph is to look at the set of live locations between each instruction and add an edge to the graph for every pair of variables in the same set. This approach is less than ideal for two reasons. First, it can be expensive because it takes $O(n^2)$ time to consider every pair in a set of n live locations. Second, in the special case where two locations hold the same value

The Racket Graph Library

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

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

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

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

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

(in-neighbors graph vertex) returns a sequence of vertices adjacent to the vertex.

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

Figure 3.6

The Racket `graph` package.

(because one was assigned to the other), they can be live at the same time without interfering with each other.

A better way to compute the interference graph is to focus on writes (Appel and Palsberg 2003). The writes performed by an instruction must not overwrite something in a live location. So for each instruction, we create an edge between the locations being written to and the live locations. (Except that a location never interferes with itself.) For the `callq` instruction, we consider all of the caller-saved registers as being written to, so an edge is added between every live variable and every caller-saved register. Also, for `movq` there is the special case of two variables holding the same value. If a live variable v is the same as the source of the `movq`, then there is no need to add an edge between v and the destination, because they both hold the same value. So we have the following two rules.

1. If instruction I_k is a move instruction of the form `movq s , d` , then for every $v \in L_{\text{after}}(k)$, if $v \neq d$ and $v \neq s$, add the edge (d, v) .
2. For any other instruction I_k , for every $d \in W(k)$ and every $v \in L_{\text{after}}(k)$, if $v \neq d$, add the edge (d, v) .

Working from the top to bottom of Figure 3.5, we apply the above rules to each instruction. We highlight a few of the instructions. The first instruction is `movq $1, v` and the live-after set is $\{v, \text{rsp}\}$. Rule 1 applies, so v interferes with `rsp`. The fourth instruction is `addq $7, x` and the live-after set is $\{w, x, \text{rsp}\}$. Rule 2 applies so x interferes with w and `rsp`. The next instruction is `movq x, y` and the live-after set is $\{w, x, y, \text{rsp}\}$. Rule 1 applies, so y interferes with w and `rsp` but not x because x is the source of the move and therefore x and y hold the same value. Figure 3.7 lists the interference results for all of the instructions and the resulting interference graph is shown in Figure 3.8.

<code>movq \$1, v</code>	<code>v</code> interferes with <code>rsp</code> ,
<code>movq \$42, w</code>	<code>w</code> interferes with <code>v</code> and <code>rsp</code> ,
<code>movq v, x</code>	<code>x</code> interferes with <code>w</code> and <code>rsp</code> ,
<code>addq \$7, x</code>	<code>x</code> interferes with <code>w</code> and <code>rsp</code> ,
<code>movq x, y</code>	<code>y</code> interferes with <code>w</code> and <code>rsp</code> but not <code>x</code> ,
<code>movq x, z</code>	<code>z</code> interferes with <code>w</code> , <code>y</code> , and <code>rsp</code> ,
<code>addq w, z</code>	<code>z</code> interferes with <code>y</code> and <code>rsp</code> ,
<code>movq y, t</code>	<code>t</code> interferes with <code>z</code> and <code>rsp</code> ,
<code>negq t</code>	<code>t</code> interferes with <code>z</code> and <code>rsp</code> ,
<code>movq z, %rax</code>	<code>rax</code> interferes with <code>t</code> and <code>rsp</code> ,
<code>addq t, %rax</code>	<code>rax</code> interferes with <code>rsp</code> .
<code>jmp conclusion</code>	no interference.

Figure 3.7

Interference results for the running example.

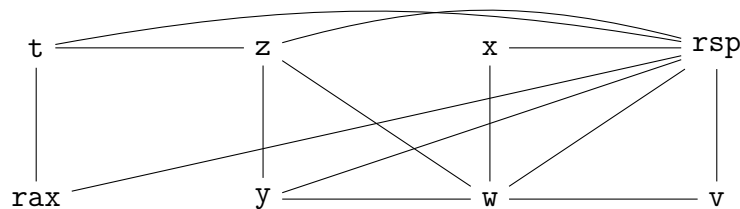


Figure 3.8

The interference graph of the example program.

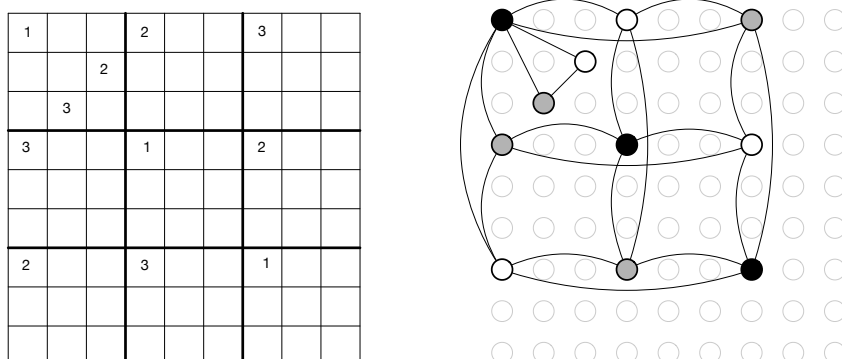
Exercise 14 Implement the compiler pass named `build_interference` according to the algorithm suggested above. We recommend using the Racket `graph` package to create and inspect the interference graph. The output graph of this pass should be stored in the `info` field of the program, under the key `conflicts`.

3.4 Graph Coloring via Sudoku

We come to the main event of this chapter, mapping variables to registers and stack locations. Variables that interfere with each other must be mapped to different locations. In terms of the interference graph, this means that adjacent vertices must be mapped to different locations. If we think of locations as colors, the register allocation problem becomes the graph coloring problem (Balakrishnan 1996; Rosen 2002).

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

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

**Figure 3.9**

A Sudoku game board and the corresponding colored graph.

- Based on the initial assignment of numbers to squares in the Sudoku board, assign the corresponding colors to the corresponding vertices in the graph.

If you can color the remaining vertices in the graph with the nine colors, then you have also solved the corresponding game of Sudoku. Figure 3.9 shows an initial Sudoku game board and the corresponding graph with colored vertices. We map the Sudoku number 1 to black, 2 to white, and 3 to gray. We only show edges for a sampling of the vertices (the colored ones) because showing edges for all of the vertices would make the graph unreadable.

Some techniques for playing Sudoku correspond to heuristics used in graph coloring algorithms. For example, one of the basic techniques for Sudoku is called Pencil Marks. The idea is to use a process of elimination to determine what numbers are no longer available 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 write the pencil mark 1 in all the squares in the same row, column, and region to indicate that 1 is no longer an option for those other squares. The Pencil Marks technique corresponds to the notion of *saturation* due to Brélaz 1979. The saturation of a vertex, in Sudoku terms, is the set of numbers that are no longer available. In graph terminology, we have the following definition:

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

where $\text{adjacent}(u)$ is the set of vertices that share an edge with u .

The Pencil Marks technique leads to a simple strategy for filling in numbers: if there is a square with only one possible number left, then choose that number! But what if there are no squares with only one possibility left? One brute-force approach is to try them all: choose the first one and if that ultimately leads to a solution, great. If not, backtrack and choose the next possibility. One good thing about Pencil Marks is that it reduces the degree of branching in the search tree. Nevertheless, backtracking can be terribly time consuming. One way to reduce the amount of

Algorithm: DSATUR

Input: a graph G

Output: an assignment $\text{color}[v]$ for each vertex $v \in G$

```

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

```

Figure 3.10

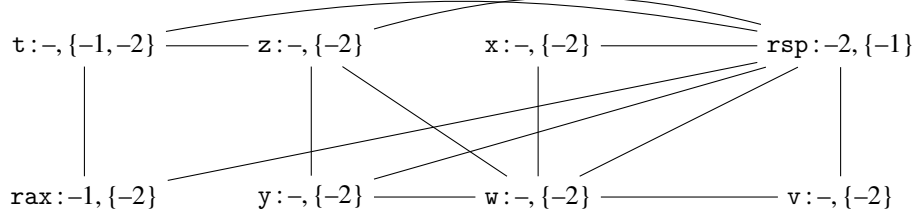
The saturation-based greedy graph coloring algorithm.

backtracking is to use the most-constrained-first heuristic (aka. minimum remaining values) (Russell and Norvig 2003). That is, when choosing a square, always choose one with the fewest possibilities left (the vertex with the highest saturation). The idea is that choosing highly constrained squares earlier rather than later is better because later on there may not be any possibilities left in the highly saturated squares.

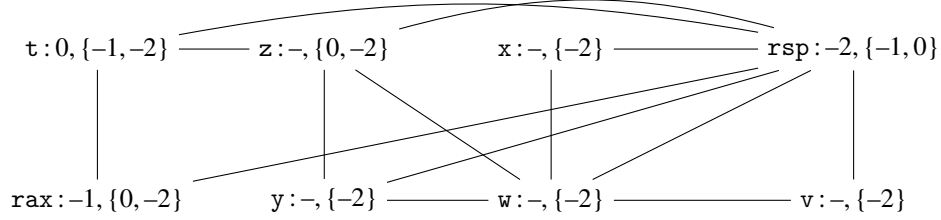
However, register allocation is easier than Sudoku because the register allocator can fall back to assigning variables to stack locations when the registers run out. Thus, it makes sense to replace backtracking with greedy search: make the best choice at the time and keep going. We still wish to minimize the number of colors needed, so we use the most-constrained-first heuristic in the greedy search. Figure 3.10 gives the pseudo-code for a simple greedy algorithm for register allocation based on saturation and the most-constrained-first heuristic. It is roughly equivalent to the DSATUR graph coloring algorithm (Brélaz 1979). Just as in Sudoku, the algorithm represents colors with integers. The integers 0 through $k-1$ correspond to the k registers that we use for register allocation. The integers k and larger correspond to stack locations. The registers that are not used for register allocation, such as **rax**, are assigned to negative integers. In particular, we assign -1 to **rax** and -2 to **rsp**.

With the DSATUR algorithm in hand, let us return to the running example and consider how to color the interference graph in Figure 3.8. We start by assigning the register nodes to their own color. For example, **rax** is assigned the color -1 and **rsp** is assigned -2 . The variables are not yet colored, so they are annotated with a dash. We then update the saturation for vertices that are adjacent to a register, obtaining the following annotated graph. For example, the saturation for **t** is $\{-1, -2\}$ because

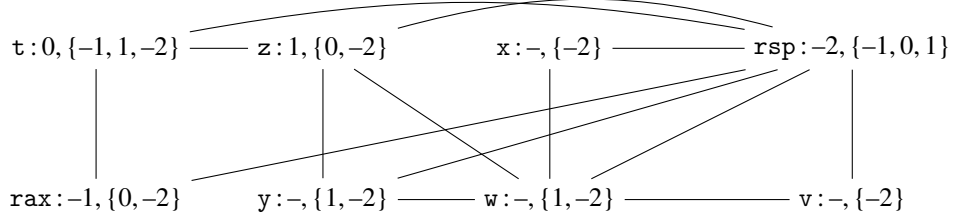
it interferes with both **rax** and **rsp**.



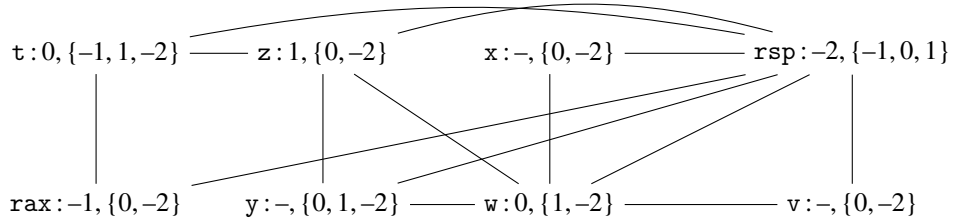
The algorithm says to select a maximally saturated vertex. So we pick **t** and color it with the first available integer, which is 0. We mark 0 as no longer available for **z**, **rax**, and **rsp** because they interfere with **t**.



We repeat the process, selecting a maximally saturated vertex, choosing is **z**, and color it with the first available number, which is 1. We add 1 to the saturation for the neighboring vertices **t**, **y**, **w**, and **rsp**.

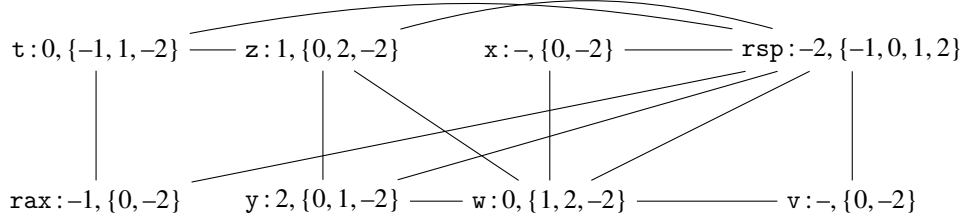


The most saturated vertices are now **w** and **y**. We color **w** with the first available color, which is 0.

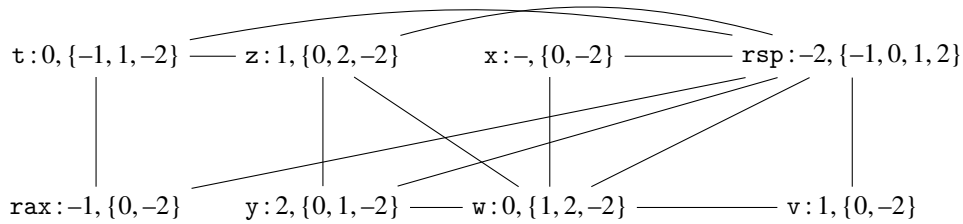


Vertex **y** is now the most highly saturated, so we color **y** with 2. We cannot choose 0 or 1 because those numbers are in **y**'s saturation set. Indeed, **y** interferes with **w**

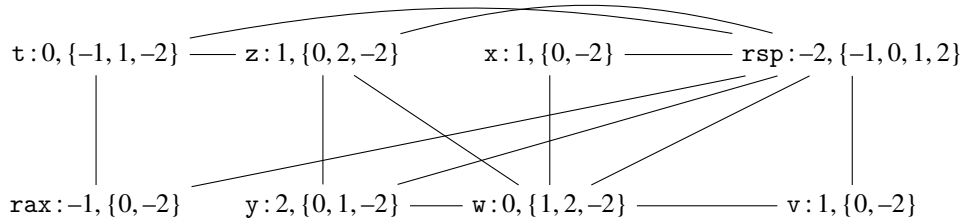
and z , whose colors are 0 and 1 respectively.



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



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



So we obtain the following coloring:

$$\{rax \mapsto -1, rsp \mapsto -2, t \mapsto 0, z \mapsto 1, x \mapsto 1, y \mapsto 2, w \mapsto 0, v \mapsto 1\}$$

We recommend creating an auxiliary function named `color_graph` that takes an interference graph and a list of all the variables in the program. This function should return a mapping of variables to their colors (represented as natural numbers). By creating this helper function, you will be able to reuse it in Chapter 7 when we add support for functions.

To prioritize the processing of highly saturated nodes inside the `color_graph` function, we recommend using the priority queue data structure described in Figure 3.11. In addition, you will need to maintain a mapping from variables to their “handles” in the priority queue so that you can notify the priority queue when their saturation changes.

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

Priority Queue

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

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

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

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

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

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

Figure 3.11

The priority queue data structure.

locations.

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

Composing this mapping with the coloring, we arrive at the following assignment of variables to locations.

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

Adapt the code from the `assign_homes` pass (Section 2.8) to replace the variables with their assigned location. Applying the above assignment to our running example, on the left, yields the program on the right.

<pre> movq \$1, v movq \$42, w movq v, x addq \$7, x movq x, y movq x, z addq w, z movq y, t negq t movq z, %rax addq t, %rax jmp conclusion </pre>	\Rightarrow	<pre> movq \$1, -8(%rbp) movq \$42, %rcx movq -8(%rbp), -8(%rbp) addq \$7, -8(%rbp) movq -8(%rbp), -16(%rbp) movq -8(%rbp), -8(%rbp) addq %rcx, -8(%rbp) movq -16(%rbp), %rcx negq %rcx movq -8(%rbp), %rax addq %rcx, %rax jmp conclusion </pre>
---	---------------	---

Exercise 15 Implement the `allocate_registers` pass. Create five programs that exercise all aspects of the register allocation algorithm, including spilling variables to the stack. Replace `assign_homes` in the list of `passes` in the `run-tests.rkt`

script with the three new passes: `uncover_live`, `build_interference`, and `allocate_registers`. Temporarily remove the call to `compiler-tests`. Run the script to test the register allocator.

3.5 Patch Instructions

The remaining step in the compilation to x86 is to ensure that the instructions have at most one argument that is a memory access. In the running example, the instruction `movq -8(%rbp), -16(%rbp)` is problematic. Recall from Section 2.9 that the fix is to first move `-8(%rbp)` into `rax` and then move `rax` into `-16(%rbp)`. The moves from `-8(%rbp)` to `-8(%rbp)` are also problematic, but they can simply be deleted. In general, we recommend deleting all the trivial moves whose source and destination are the same location. The following is the output of `patch_instructions` on the running example.

<pre> movq \$1, -8(%rbp) movq \$42, %rcx movq -8(%rbp), -8(%rbp) addq \$7, -8(%rbp) movq -8(%rbp), -16(%rbp) movq -8(%rbp), -8(%rbp) addq %rcx, -8(%rbp) movq -16(%rbp), %rcx negq %rcx movq -8(%rbp), %rax addq %rcx, %rax jmp conclusion </pre>	\Rightarrow	<pre> movq \$1, -8(%rbp) movq \$42, %rcx addq \$7, -8(%rbp) movq -8(%rbp), %rax movq %rax, -16(%rbp) addq %rcx, -8(%rbp) movq -16(%rbp), %rcx negq %rcx movq -8(%rbp), %rax addq %rcx, %rax jmp conclusion </pre>
---	---------------	---

Exercise 16 Update the `patch_instructions` compiler pass to delete trivial moves. Run the script to test the `patch_instructions` pass.

3.6 Prelude and Conclusion

Recall that this pass generates the prelude and conclusion instructions to satisfy the x86 calling conventions (Section 3.1). With the addition of the register allocator, the callee-saved registers used by the register allocator must be saved in the prelude and restored in the conclusion. In the `allocate_registers` pass, add an entry to the *info* of `X86Program` named `used_callee` that stores the set of callee-saved registers that were assigned to variables. The `prelude_and_conclusion` pass can then access this information to decide which callee-saved registers need to be saved and restored. When calculating the amount to adjust the `rsp` in the prelude, make sure to take into account the space used for saving the callee-saved registers. Also, don't forget that the frame needs to be a multiple of 16 bytes! We recommend using the following equation for the amount A to subtract from the `rsp`. Let S be the number of spilled variables and C be the number of callee-saved registers that were allocated to variables. The *align* function rounds a number up to the nearest

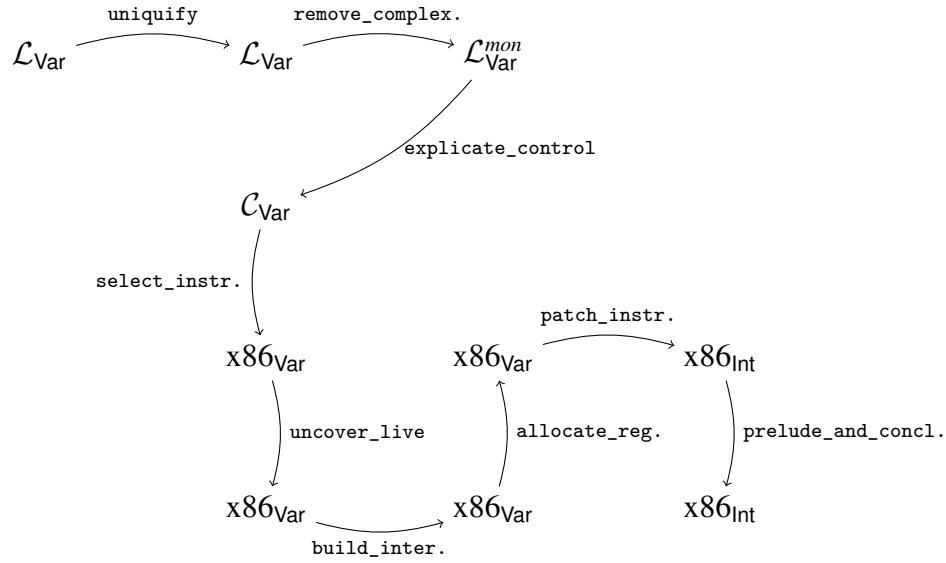
**Figure 3.12**

Diagram of the passes for \mathcal{L}_{Var} with register allocation.

16 bytes.

$$A = \text{align}(8S + 8C) - 8C$$

The reason we subtract $8C$ in the above equation is because the prelude uses `pushq` to save each of the callee-saved registers, and `pushq` subtracts 8 from the `rsp`.

An overview of all of the passes involved in register allocation is shown in Figure 3.12.

Figure 3.13 shows the x86 code generated for the running example (Figure 3.1). To demonstrate both the use of registers and the stack, we limit the register allocator for this example to use just two registers: `rbx` and `rcx`. In the prelude of the `main` function, we push `rbx` onto the stack because it is a callee-saved register and it was assigned to a variable by the register allocator. We subtract 8 from the `rsp` at the end of the prelude to reserve space for the one spilled variable. After that subtraction, the `rsp` is aligned to 16 bytes.

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

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

```

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

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

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

```

Figure 3.13

The x86 output from the running example (Figure 3.1), limiting allocation to just `rbx` and `rcx`.

Exercise 17 Update the `prelude_and_conclusion` pass as described in this section. In the `run-tests.rkt` script, add `prelude_and_conclusion` to the list of passes and the call to `compiler-tests`. Run the script to test the complete compiler for \mathcal{L}_{Var} that performs register allocation.

3.7 Challenge: Move Biasing

This section describes an enhancement to the register allocator, called move biasing, for students who are looking for an extra challenge.

To motivate the need for move biasing we return to the running example but this time we use all of the general purpose registers. So we have the following mapping of color numbers to registers.

$$\{0 \mapsto \%rcx, 1 \mapsto \%rdx, 2 \mapsto \%rsi, \dots\}$$

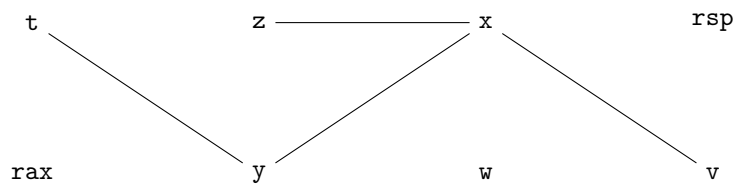
Using the same assignment of variables to color numbers that was produced by the register allocator described in the last section, we get the following program.

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

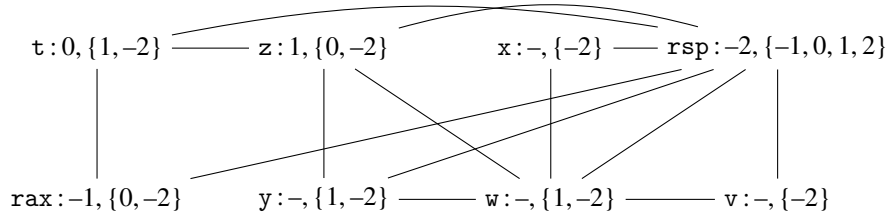
In the above output code there are two `movq` instructions that can be removed because their source and target are the same. However, if we had put `t`, `v`, `x`, and `y` into the same register, we could instead remove three `movq` instructions. We can accomplish this by taking into account which variables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

We say that two variables p and q are *move related* if they participate together in a `movq` instruction, that is, `movq p, q` or `movq q, p`. When deciding which variable to color next, when there are multiple variables with the same saturation, prefer variables that can be assigned to a color that is the same as the color of a move-related variable. Furthermore, 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. So this preference should be used as a tie breaker when choosing between registers or when choosing between stack locations.

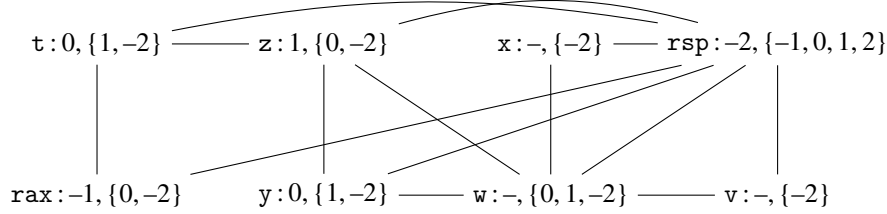
We recommend representing the move relationships in a graph, similar to how we represented interference. The following is the *move graph* for our running example.



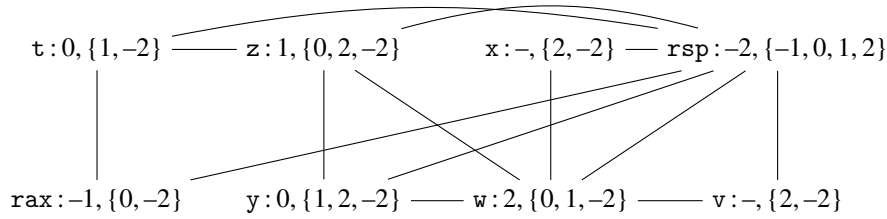
Now we replay the graph coloring, pausing to see the coloring of `y`. Recall the following configuration. The most saturated vertices were `w` and `y`.



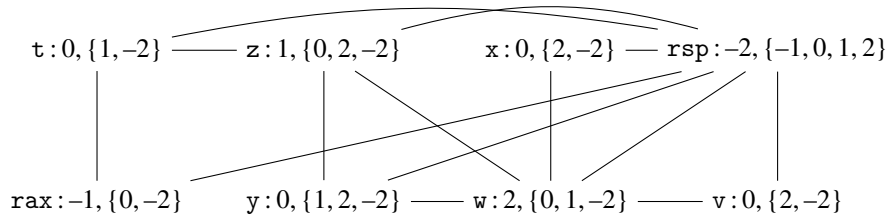
Last time we chose to color w with 0. But this time we see that w is not move related to any vertex, but y is move related to t . So we choose to color y with 0, the same color as t .



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



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



So we have the following assignment of variables to registers.

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

We apply this register assignment to the running example, on the left, to obtain the code in the middle. The `patch_instructions` then deletes the trivial moves to obtain the code on the right.

<pre> movq \$1, v movq \$42, w movq v, x addq \$7, x movq x, y movq x, z addq w, z movq y, t negq t movq z, %rax addq t, %rax jmp conclusion </pre>	⇒	<pre> movq \$1, %rcx movq \$42, %rsi movq %rcx, %rcx addq \$7, %rcx movq %rcx, %rcx movq %rcx, %rdx addq %rsi, %rdx movq %rcx, %rcx negq %rcx movq %rdx, %rax addq %rcx, %rax jmp conclusion </pre>	⇒	<pre> movq \$1, %rcx movq \$42, %rsi addq \$7, %rcx movq %rcx, %rdx addq %rsi, %rdx negq %rcx movq %rdx, %rax addq %rcx, %rax jmp conclusion </pre>
---	---	---	---	---

Exercise 18 Change your implementation of `allocate_registers` to take move biasing into account. 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. Make sure that your compiler still passes all of the tests.

3.8 Further Reading

Early register allocation algorithms were developed for Fortran compilers in the 1950s (Horwitz et al. 1966; Backus 1978). The use of graph coloring began in the late 1970s and early 1980s with the work of Chaitin et al. (1981) on an optimizing compiler for PL/I. The algorithm is based on the following observation of Kempe (1879). If a graph G has a vertex v with degree lower than k , then G is k colorable if the subgraph of G with v removed is also k colorable. To see why, suppose that the subgraph is k colorable. At worst the neighbors of v are assigned different colors, but since there are less than k neighbors, there will be one or more colors left over to use for coloring v in G .

The algorithm of Chaitin et al. (1981) removes a vertex v of degree less than k from the graph and recursively colors the rest of the graph. Upon returning from the recursion, it colors v with one of the available colors and returns. Chaitin (1982) augments this algorithm to handle spilling as follows. If there are no vertices of degree lower than k then pick a vertex at random, spill it, remove it from the graph, and proceed recursively to color the rest of the graph.

Prior to coloring, Chaitin et al. (1981) merge variables that are move-related and that don't interfere with each other, a process called *coalescing*. While coalescing decreases the number of moves, it can make the graph more difficult to color. Briggs, Cooper, and Torczon (1994) propose *conservative coalescing* in which two variables are merged only if they have fewer than k neighbors of high degree. George and Appel (1996) observe that conservative coalescing is sometimes too conservative and make it more aggressive by iterating the coalescing with the removal of low-degree vertices. Attacking the problem from a different angle, Briggs, Cooper, and Torczon (1994) also propose *biased coloring* in which a variable is assigned to the same color as another move-related variable if possible, as discussed in Section 3.7. The

algorithm of Chaitin et al. (1981) and its successors iteratively performs coalescing, graph coloring, and spill code insertion until all variables have been assigned a location.

Briggs, Cooper, and Torczon (1994) observes that Chaitin (1982) sometimes spills variables that don't have to be: a high-degree variable can be colorable if many of its neighbors are assigned the same color. Briggs, Cooper, and Torczon (1994) propose *optimistic coloring*, in which a high-degree vertex is not immediately spilled. Instead the decision is deferred until after the recursive call, at which point it is apparent whether there is actually an available color or not. We observe that this algorithm is equivalent to the smallest-last ordering algorithm (Matula, Marble, and Isaacson 1972) if one takes the first k colors to be registers and the rest to be stack locations. Earlier editions of the compiler course at Indiana University (Dybvig and Keep 2010) were based on the algorithm of Briggs, Cooper, and Torczon (1994).

The smallest-last ordering algorithm is one of many *greedy* coloring algorithms. A greedy coloring algorithm visits all the vertices in a particular order and assigns each one the first available color. An *offline* greedy algorithm chooses the ordering up-front, prior to assigning colors. The algorithm of Chaitin et al. (1981) should be considered offline because the vertex ordering does not depend on the colors assigned. Other orderings are possible. For example, Chow and Hennessy (1984) order variables according to an estimate of runtime cost.

An *online* greedy coloring algorithm uses information about the current assignment of colors to influence the order in which the remaining vertices are colored. The saturation-based algorithm described in this chapter is one such algorithm. We choose to use saturation-based coloring because it is fun to introduce graph coloring via Sudoku!

A register allocator may choose to map each variable to just one location, as in Chaitin et al. (1981), or it may choose to map a variable to one or more locations. The later can be achieved by *live range splitting*, where a variable is replaced by several variables that each handle part of its live range (Chow and Hennessy 1984; Briggs, Cooper, and Torczon 1994; Cooper and Simpson 1998).

Palsberg (2007) observe that many of the interference graphs that arise from Java programs in the JoeQ compiler are *chordal*, that is, every cycle with four or more edges has an edge which is not part of the cycle but which connects two vertices on the cycle. Such graphs can be optimally colored by the greedy algorithm with a vertex ordering determined by maximum cardinality search.

In situations where compile time is of utmost importance, such as in just-in-time compilers, graph coloring algorithms can be too expensive and the linear scan algorithm of Poletto and Sarkar (1999) may be more appropriate.

4 Booleans and Conditionals

The \mathcal{L}_{Var} language only has a single kind of value, the integers. In this chapter we add a second kind of value, the Booleans, to create the \mathcal{L}_{If} language. The Boolean values *true* and *false* are written `#t` and `#f` respectively in Racket. The \mathcal{L}_{If} language includes several operations that involve Booleans (`and`, `not`, `eq?`, `<`, etc.) and the `if` expression. With the addition of `if`, programs can have non-trivial control flow which impacts `explicate_control` and liveness analysis. Also, because we now have two kinds of values, we need to handle programs that apply an operation to the wrong kind of value, such as `(not 1)`.

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

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

The \mathcal{L}_{If} language performs type checking during compilation like Typed Racket. In Chapter 9 we study the alternative choice, that is, a dynamically typed language like Racket. The \mathcal{L}_{If} language is a subset of Typed Racket; for some operations we are more restrictive, for example, rejecting `(not 1)`. We keep the type checker for \mathcal{L}_{If} fairly simple because the focus of this book is on compilation, not type systems, about which there are already several excellent books (Pierce 2002, 2004; Harper 2016; Pierce et al. 2018).

This chapter is organized as follows. We begin by defining the syntax and interpreter for the \mathcal{L}_{If} language (Section 4.1). We then introduce the idea of type checking and define a type checker for \mathcal{L}_{If} (Section 4.2). To compile \mathcal{L}_{If} we need to enlarge the intermediate language \mathcal{C}_{Var} into \mathcal{C}_{If} (Section 4.3) and `x86Int` into `x86If` (Section 4.4). The remaining sections of this chapter discuss how Booleans and conditional control flow require changes to the existing compiler passes and the addition of new ones. We introduce the `shrink` pass to translate some operators into others, thereby reducing the number of operators that need to be handled in later passes. The

$type$	$::=$	$Integer$
exp	$::=$	$int \mid (read) \mid (-\ exp) \mid (+\ exp\ exp) \mid (-\ exp\ exp)$
exp	$::=$	$var \mid (let\ ([var\ exp])\ exp)$
$type$	$::=$	$Boolean$
$bool$	$::=$	$\#t \mid \#f$
cmp	$::=$	$eq? \mid < \mid <= \mid > \mid >=$
exp	$::=$	$bool \mid (and\ exp\ exp) \mid (or\ exp\ exp) \mid (not\ exp)$
		$\mid (cmp\ exp\ exp) \mid (if\ exp\ exp\ exp)$
\mathcal{L}_{if}	$::=$	exp

Figure 4.1

The concrete syntax of \mathcal{L}_{if} , extending \mathcal{L}_{var} (Figure 2.1) with Booleans and conditionals.

main event of this chapter is the `explicate_control` pass that is responsible for translating `if`'s into conditional `goto`'s (Section 4.8). Regarding register allocation, there is the interesting question of how to handle conditional `goto`'s during liveness analysis.

4.1 The \mathcal{L}_{if} Language

The concrete and abstract syntax of the \mathcal{L}_{if} language are defined in Figures 4.1 and 4.2, respectively. The \mathcal{L}_{if} language includes all of \mathcal{L}_{var} (shown in gray), the Boolean literals `#t` and `#f`, and the `if` expression. We expand the set of operators to include

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

We reorganize the abstract syntax for the primitive operations in Figure 4.2, using only one grammar rule for all of them. This means that the grammar no longer checks whether the arity of an operators matches the number of arguments. That responsibility is moved to the type checker for \mathcal{L}_{if} (Section 4.2).

Figure 4.3 defines the interpreter for \mathcal{L}_{if} , which inherits from the interpreter for \mathcal{L}_{var} (Figure 2.5). The literals `#t` and `#f` evaluate to the corresponding Boolean values. The conditional expression `(if e_1 e_2 e_3)` evaluates expression e_1 and then either evaluates e_2 or e_3 depending on whether e_1 produced `#t` or `#f`. The logical operations `and`, `or`, and `not` behave according to propositional logic. In addition, the `and` and `or` operations perform *short-circuit evaluation*. That is, given the expression `(and e_1 e_2)`, the expression e_2 is not evaluated if e_1 evaluates to `#f`. Similarly, given `(or e_1 e_2)`, the expression e_2 is not evaluated if e_1 evaluates to `#t`.

With the increase in the number of primitive operations, the interpreter would become repetitive without some care. We refactor the case for `Prim`, moving the code that differs with each operation into the `interp_op` method shown in in Figure 4.4.

<i>type</i>	::=	Integer
<i>op</i>	::=	read + -
<i>exp</i>	::=	(Int <i>int</i>) (Prim <i>op</i> (<i>exp</i> ...))
<i>exp</i>	::=	(Var <i>var</i>) (Let <i>var exp exp</i>)
<i>type</i>	::=	Boolean
<i>bool</i>	::=	#t #f
<i>cmp</i>	::=	eq? < <= > >=
<i>op</i>	::=	<i>cmp</i> and or not
<i>exp</i>	::=	(Bool <i>bool</i>) (If <i>exp exp exp</i>)
\mathcal{L}_{If}	::=	(Program '() <i>exp</i>)

Figure 4.2

The abstract syntax of \mathcal{L}_{If} .

We handle the **and** and **or** operations separately because of their short-circuiting behavior.

4.2 Type Checking \mathcal{L}_{If} Programs

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

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

produces a value of type **Integer** while

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

produces a value of type **Boolean**.

A second way to think about type checking is that it enforces a set of rules about which operators can be applied to which kinds of values. For example, our type checker for \mathcal{L}_{If} signals an error for the below expression

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

The subexpression `(+ 10 (- (+ 12 20)))` has type **Integer** but the type checker enforces the rule that the argument of **not** must be an expression of type **Boolean**.

We implement type checking using classes and methods because they provide the open recursion needed to reuse code as we extend the type checker in later chapters, analogous to the use of classes and methods for the interpreters (Section 2.1.1).

We separate the type checker for the \mathcal{L}_{Var} subset into its own class, shown in Figure 4.5. The type checker for \mathcal{L}_{If} is shown in Figure 4.6 and it inherits from the type checker for \mathcal{L}_{Var} . These type checkers are in the files `type-check-Lvar.rkt` and `type-check-Lif.rkt` of the support code. Each type checker is a structurally recursive function over the AST. Given an input expression *e*, the type checker

```

(define interp-Lif-class
  (class interp-Lvar-class
    (super-new)

    (define/public (interp_op op) ...)

    (define/override ((interp_exp env) e)
      (define recur (interp_exp env))
      (match e
        [(Bool b) b]
        [(If cnd thn els)
         (match (recur cnd)
           [#t (recur thn)]
           [#f (recur els)])]
        [(Prim 'and (list e1 e2))
         (match (recur e1)
           [#t (match (recur e2) [#t #t] [#f #f])]
           [#f #f])]
        [(Prim 'or (list e1 e2))
         (define v1 (recur e1))
         (match v1
           [#t #t]
           [#f (match (recur e2) [#t #t] [#f #f])])]
        [(Prim op args)
         (apply (interp_op op) (for/list ([e args]) (recur e)))]
        [else ((super interp_exp env) e)])
      ))

    (define (interp_Lif p)
      (send (new interp-Lif-class) interp_program p))

```

Figure 4.3

Interpreter for the \mathcal{L}_{ff} language. (See Figure 4.4 for `interp-op`.)

either signals an error or returns an expression and its type. It returns an expression because there are situations in which we want to change or update the expression.

Next we discuss the `type_check_exp` function of \mathcal{L}_{Var} in Figure 4.5. The type of an integer constant is `Integer`. To handle variables, the type checker uses the environment `env` to map variables to types. Consider the case for `let`. We type check the initializing expression to obtain its type `T` and then associate type `T` with the variable `x` in the environment used to type check the body of the `let`. Thus, when the type checker encounters a use of variable `x`, it can find its type in the environment. Regarding primitive operators, we recursively analyze the arguments and then invoke `type_check_op` to check whether the argument types are allowed.

Several auxiliary methods are used in the type checker. The method `operator-types` defines a dictionary that maps the operator names to their parameter and return types. The `type-equal?` method determines whether two types are equal, which for now simply dispatches to `equal?` (deep equality). The `check-type-equal?` method triggers an error if the two types are not equal. The

```

(define/public (interp_op op)
  (match op
    ['+ fx+]
    ['- fx-]
    ['read read-fixnum]
    ['not (lambda (v) (match v [#t #f] [#f #t]))]
    ['eq? (lambda (v1 v2)
      (cond [(or (and (fixnum? v1) (fixnum? v2))
                 (and (boolean? v1) (boolean? v2))
                 (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")]))

```

Figure 4.4

Interpreter for the primitive operators in the \mathcal{L}_{if} language.

`type-check-op` method looks up the operator in the `operator-types` dictionary and then checks whether the argument types are equal to the parameter types. The result is the return type of the operator.

The type checker for \mathcal{L}_{if} is defined in Figure 4.6. The type of a Boolean constant is `Boolean`. The `operator-types` function adds dictionary entries for the new operators. The equality operator requires the two arguments to have the same type and therefore we handle it separately from the other operators. The condition of an if must be of `Boolean` type and the two branches must have the same type.

Exercise 19 Create 10 new test programs in \mathcal{L}_{if} . Half of the programs should have a type error. For those programs, create an empty file with the same base name but with file extension `.tyerr`. For example, if the test `cond_test_14.rkt` is expected to error, then create an empty file named `cond_test_14.tyerr`. This indicates to `interp-tests` and `compiler-tests` that a type error is expected. The other half of the test programs should not have type errors. In the `run-tests.rkt` script, change the second argument of `interp-tests` and `compiler-tests` to `type-check-Lif`, which causes the type checker to run prior to the compiler passes. Temporarily change the `passes` to an empty list and run the script, thereby checking that the new test programs either type check or not as intended. Run the test script to check that these test programs type check as expected.

```

(define type-check-Lvar-class
  (class object%
    (super-new)

    (define/public (operator-types)
      '((+ . ((Integer Integer) . Integer))
        (- . ((Integer Integer) . Integer))
        (read . (() . Integer))))

    (define/public (type-equal? t1 t2) (equal? t1 t2))

    (define/public (check-type-equal? t1 t2 e)
      (unless (type-equal? t1 t2)
        (error 'type-check "~a != ~a\nin ~v" t1 t2 e)))

    (define/public (type-check-op op arg-types e)
      (match (dict-ref (operator-types) op)
        [`,param-types . ,return-type
         (for ([at arg-types] [pt param-types])
           (check-type-equal? at pt e))
         return-type]
        [else (error 'type-check-op "unrecognized ~a" op)]))

    (define/public (type-check-exp env)
      (lambda (e)
        (match e
          [(Int n) (values (Int n) 'Integer)]
          [(Var x) (values (Var x) (dict-ref env x))]
          [(Let x e body)
           (define-values (e~ Te) ((type-check-exp env) e))
           (define-values (b Tb) ((type-check-exp (dict-set env x Te)) body))
           (values (Let x e~ b) Tb)]
          [(Prim op es)
           (define-values (new-es ts)
             (for/lists (exprs types) ([e es]) ((type-check-exp env) e)))
           (values (Prim op new-es) (type-check-op op ts e))]
          [else (error 'type-check-exp "couldn't match" e)])))

    (define/public (type-check-program e)
      (match e
        [(Program info body)
         (define-values (body~ Tb) ((type-check-exp '()) body))
         (check-type-equal? Tb 'Integer body)
         (Program info body~)]
        [else (error 'type-check-Lvar "couldn't match ~a" e)]))
  ))

(define (type-check-Lvar p)
  (send (new type-check-Lvar-class) type-check-program p))

```

Figure 4.5

Type checker for the \mathcal{L}_{Var} language.


```

(define type-check-Lif-class
  (class type-check-Lvar-class
    (super-new)
    (inherit check-type-equal?)

    (define/override (operator-types)
      (append '((and . ((Boolean Boolean) . Boolean))
                (or . ((Boolean Boolean) . Boolean))
                (< . ((Integer Integer) . Boolean))
                (<= . ((Integer Integer) . Boolean))
                (> . ((Integer Integer) . Boolean))
                (>= . ((Integer Integer) . Boolean))
                (not . ((Boolean) . Boolean)))
              (super operator-types)))

    (define/override (type-check-exp env)
      (lambda (e)
        (match e
          [(Bool b) (values (Bool b) 'Boolean)]
          [(Prim 'eq? (list e1 e2))
           (define-values (e1^ T1) ((type-check-exp env) e1))
           (define-values (e2^ T2) ((type-check-exp env) e2))
           (check-type-equal? T1 T2 e)
           (values (Prim 'eq? (list e1^ e2^)) 'Boolean)]
          [(If cnd thn els)
           (define-values (cnd^ Tc) ((type-check-exp env) cnd))
           (define-values (thn^ Tt) ((type-check-exp env) thn))
           (define-values (els^ Te) ((type-check-exp env) els))
           (check-type-equal? Tc 'Boolean e)
           (check-type-equal? Tt Te e)
           (values (If cnd^ thn^ els^ Te))]
          [else ((super type-check-exp env) e)])))
      ))

    (define (type-check-Lif p)
      (send (new type-check-Lif-class) type-check-program p))

```

Figure 4.6

Type checker for the \mathcal{L}_{If} language.

```

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

```

Figure 4.7

The concrete syntax of the C_{lf} intermediate language, an extension of C_{var} (Figure 2.12).

```

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

```

Figure 4.8

The abstract syntax of C_{lf} , an extension of C_{var} (Figure 2.13).

4.3 The C_{lf} Intermediate Language

The C_{lf} language builds on C_{var} by adding logical and comparison operators to the *exp* non-terminal and the literals **#t** and **#f** to the *arg* non-terminal. Regarding control flow, C_{lf} adds **goto** and **if** statements to the *tail* non-terminal. The condition of an **if** statement is a comparison operation and the branches are **goto** statements, making it straightforward to compile **if** statements to x86. The **CProgram** construct contains an alist mapping labels to *tail* expressions. A **goto** statement transfers control to the *tail* expression corresponding to its label. Figure 4.7 defines the concrete syntax of the C_{lf} intermediate language and Figure 4.8 defines its abstract syntax.

4.4 The x86_{lf} Language

To implement the new logical operations, the comparison operations, and the **if** expression, we delve further into the x86 language. Figures 4.9 and 4.10 define the concrete and abstract syntax for the x86_{lf} subset of x86, which includes instructions for logical operations, comparisons, and conditional jumps.

```

    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 |
             pushq arg | popq arg | callq label | retq | jmp label |
             label: instr
    -----
    bytereg ::= ah | al | bh | bl | ch | cl | dh | dl
    arg      ::= %bytereg
    cc       ::= e | ne | l | le | g | ge
    instr    ::= xorq arg, arg | cmpq arg, arg | setcc arg | movzbq arg, arg
             | jcc label
    x86lf ::= .globl main
             main: instr ...

```

Figure 4.9

The concrete syntax of $x86_{lf}$ (extends $x86_{int}$ of Figure 2.6).

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

	0	1
0	0	1
1	1	0

For example, applying XOR to each bit of the binary numbers 0011 and 0101 yields 0110. Notice that in the row of the table for the bit 1, the result is the opposite of the second bit. Thus, the **not** operation can be implemented by **xorq** with 1 as the first argument as follows, where *arg* is the translation of *atm* to x86.

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

Next we consider the x86 instructions that are relevant for compiling the comparison operations. The **cmpq** instruction compares its two arguments to determine whether one argument is less than, equal, or greater than the other argument. The **cmpq** instruction is unusual regarding the order of its arguments and where the result is placed. The argument order is backwards: if you want to test whether $x < y$, then write **cmpq** *y*, *x*. The result of **cmpq** is placed in the special EFLAGS register. This register cannot be accessed directly but it can be queried by a number of instructions, including the **set** instruction. The instruction **setcc** *d* puts a 1 or 0 into the destination *d* depending on whether the contents of the EFLAGS register matches the condition code *cc*: **e** for equal, **l** for less, **le** for less-or-equal,

```

reg    ::=  rsp | rbp | rax | rbx | rcx | rdx | rsi | rdi |
           r8 | r9 | r10 | r11 | r12 | r13 | r14 | r15
arg    ::=  (Imm int) | (Reg reg) | (Deref reg int)
instr  ::=  (Instr addq (arg arg)) | (Instr subq (arg arg)) | (Instr negq (arg))
           | (Instr movq (arg arg)) | (Instr pushq (arg)) | (Instr popq (arg))
           | (Callq label int) | (Retq) | (Jmp label)
block  ::=  (Block info (instr ...))
-----
bytereg ::= ah | al | bh | bl | ch | cl | dh | dl
arg      ::= (ByteReg bytereg)
cc       ::= e | l | le | g | ge
instr    ::= (Instr xorq (arg arg)) | (Instr cmpq (arg arg))
           | (Instr set (cc arg)) | (Instr movzbq (arg arg))
           | (JmpIf 'cc' label)
x86lf ::= (X86Program info ((label . block) ...))

```

Figure 4.10

The abstract syntax of $x86_{lf}$ (extends $x86_{int}$ of Figure 2.10).

g for greater, **ge** for greater-or-equal. The **set** instruction has a quirk in that its destination argument must be single byte register, such as **al** (L for lower bits) or **ah** (H for higher bits), which are part of the **rax** register. Thankfully, the **movzbq** instruction can be used to move from a single byte register to a normal 64-bit register. The abstract syntax for the **set** instruction differs from the concrete syntax in that it separates the instruction name from the condition code.

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

4.5 Shrink the \mathcal{L}_{lf} Language

The \mathcal{L}_{lf} language includes several features that are easily expressible with other features. For example, **and** and **or** are expressible using **if** as follows.

$$(\text{and } e_1 \ e_2) \Rightarrow (\text{if } e_1 \ e_2 \ \#f)$$

$$(\text{or } e_1 \ e_2) \Rightarrow (\text{if } e_1 \ \#t \ e_2)$$

By performing these translations in the front-end of the compiler, subsequent passes of the compiler do not need to deal with these features, making the passes shorter.

On the other hand, sometimes translations reduce the efficiency of the generated code by increasing the number of instructions. For example, expressing subtraction in terms of negation

$$(- e_1 e_2) \Rightarrow (+ e_1 (- e_2))$$

produces code with two x86 instructions (`negq` and `addq`) instead of just one (`subq`).

Exercise 20 Implement the pass `shrink` to remove `and` and `or` from the language by translating them to `if` expressions in \mathcal{L}_{if} . Create four test programs that involve these operators. In the `run-tests.rkt` script, add the following entry for `shrink` to the list of passes (it should be the only pass at this point).

```
(list "shrink" shrink interp_Lif type-check-Lif)
```

This instructs `interp-tests` to run the interpreter `interp_Lif` and the type checker `type-check-Lif` on the output of `shrink`. Run the script to test your compiler on all the test programs.

4.6 Uniquify Variables

Add cases to `uniquify_exp` to handle Boolean constants and `if` expressions.

Exercise 21 Update the `uniquify_exp` for \mathcal{L}_{if} and add the following entry to the list of passes in the `run-tests.rkt` script.

```
(list "uniquify" uniquify interp_Lif type-check-Lif)
```

Run the script to test your compiler.

4.7 Remove Complex Operands

The output language of `remove_complex_operands` is $\mathcal{L}_{\text{if}}^{\text{mon}}$ (Figure 4.11), the monadic normal form of \mathcal{L}_{if} . A Boolean constant is an atomic expressions but the `if` expression is not. All three sub-expressions of an `if` are allowed to be complex expressions but the operands of `not` and the comparisons must be atomic.

Add cases to the `rco_exp` and `rco_atom` functions for the new features in \mathcal{L}_{if} . When recursively processing subexpressions, recall that you should invoke `rco_atom` when the output needs to be an *atom* (as specified in the grammar for $\mathcal{L}_{\text{if}}^{\text{mon}}$) and invoke `rco_exp` when the output should be *exp*. Regarding `if`, it is particularly important to **not** replace its condition with a temporary variable because that would interfere with the generation of high-quality output in the upcoming `explicate_control` pass.

Exercise 22 Add cases for Boolean constants and `if` to the `rco_atom` and `rco_exp` functions in `compiler.rkt`. Create three new \mathcal{L}_{if} programs that exercise the interesting code in this pass. In the `run-tests.rkt` script, add the following entry to the list of passes and then run the script to test your compiler.

```
(list "remove-complex" remove_complex_operands interp_Lif type-check-Lif)
```

$ \begin{aligned} atm &::= (Int\ int) \mid (Var\ var) \\ exp &::= atm \mid (Prim\ 'read\ ()) \\ &\quad \mid (Prim\ '-\ (atm)) \mid (Prim\ '+\ (atm\ atm)) \mid (Prim\ '-\ (atm\ atm)) \\ &\quad \mid (Let\ var\ exp\ exp) \\ \hline atm &::= (Bool\ bool) \\ exp &::= (Prim\ not\ (atm)) \mid (Prim\ cmp\ (atm\ atm)) \mid (If\ exp\ exp\ exp) \\ \mathcal{L}_{if}^{mon} &::= (Program\ ()\ exp) \end{aligned} $
--

Figure 4.11

\mathcal{L}_{if}^{mon} is \mathcal{L}_{if} in monadic normal form (extends \mathcal{L}_{Var}^{mon} in Figure 2.15).

4.8 Explicate Control

Recall that the purpose of `explicate_control` is to make the order of evaluation explicit in the syntax of the program. With the addition of `if` this gets more interesting. The `explicate_control` pass translates from \mathcal{L}_{if} to \mathcal{C}_{if} . The main challenge to overcome is that the condition of an `if` can be an arbitrary expression in \mathcal{L}_{if} whereas in \mathcal{C}_{if} the condition must be a comparison.

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

```

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

```

The naive way to compile `if` and the comparison operations would be to handle each of them in isolation, regardless of their context. Each comparison would be translated into a `cmpq` instruction followed by several instructions to move the result from the EFLAGS register into a general purpose register or stack location. Each `if` would be translated into a `cmpq` instruction followed by a conditional jump. The generated code for the inner `if` in the above example would be as follows.

```

cmpq $1, x
setl %al
movzbq %al, tmp
cmpq $1, tmp
je then_branch_1
jmp else_branch_1

```

Notice that the three instructions starting with `setl` are redundant: the conditional jump could come immediately after the first `cmpq`.

Our goal will be to compile `if` expressions so that the relevant comparison instruction appears directly before the conditional jump. For example, we want to generate the following code for the inner `if`.

```
cmpq $1, x
jl then_branch_1
jmp else_branch_1
```

One way to achieve this goal is to reorganize the code at the level of \mathcal{L}_{If} , pushing the outer `if` inside the inner one, yielding the following code.

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

Unfortunately, this approach duplicates the two branches from the outer `if` and a compiler must never duplicate code! After all, the two branches could be very large expressions.

How can we apply the above transformation but without duplicating code? In other words, how can two different parts of a program refer to one piece of code. The answer is that we must move away from abstract syntax *trees* and instead use *graphs*. At the level of x86 assembly this is straightforward because we can label the code for each branch and insert jumps in all the places that need to execute the branch. In this way, jump instructions are edges in the graph and the basic blocks are the nodes. Likewise, our language \mathcal{C}_{If} provides the ability to label a sequence of statements and to jump to a label via `goto`.

As a preview of what `explicate_control` will do, Figure 4.12 shows the output of `explicate_control` on the above example. Note how the condition of every `if` is a comparison operation and that we have not duplicated any code, but instead used labels and `goto` to enable sharing of code.

Recall that in Section 2.6 we implement `explicate_control` for \mathcal{L}_{Var} using two recursive functions, `explicate_tail` and `explicate_assign`. The former function translates expressions in tail position whereas the later function translates expressions on the right-hand-side of a `let`. With the addition of `if` expression to \mathcal{L}_{If} we have a new kind of position to deal with: the predicate position of the `if`. We need another function, `explicate_pred`, that decides how to compile an `if` by analyzing its condition. So `explicate_pred` takes an \mathcal{L}_{If} expression and two \mathcal{C}_{If} tails for the then-branch and else-branch and outputs a tail. In the following paragraphs we discuss specific cases in the `explicate_tail`, `explicate_assign`, and `explicate_pred` functions.

```

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

⇒

start:
  x = (read);
  y = (read);
  if (< x 1)
    goto block_4;
  else
    goto block_5;
block_4:
  if (eq? x 0)
    goto block_2;
  else
    goto block_3;
block_5:
  if (eq? x 2)
    goto block_2;
  else
    goto block_3;
block_2:
  return (+ y 2);
block_3:
  return (+ y 10);

```

Figure 4.12

Translation from \mathcal{L}_{if} to \mathcal{C}_{if} via the `explicate_control`.

4.8.1 Explicate Tail and Assign

The `explicate_tail` and `explicate_assign` functions need additional cases for Boolean constants and `if`. The cases for `if` should recursively compile the two branches using either `explicate_tail` or `explicate_assign`, respectively. The cases should then invoke `explicate_pred` on the condition expression, passing in the generated code for the two branches. For example, consider the following program with an `if` in tail position.

```

(let ([x (read)])
  (if (eq? x 0) 42 777))

```

The two branches are recursively compiled to return statements. We then delegate to `explicate_pred`, passing the condition `(eq? x 0)` and the two return statements. We return to this example shortly when we discuss `explicate_pred`.

Next let us consider a program with an `if` on the right-hand side of a `let`.

```

(let ([y (read)])
  (let ([x (if (eq? y 0) 40 777)])
    (+ x 2)))

```

Note that the body of the inner `let` will have already been compiled to `return (+ x 2)`; and passed as the `cont` parameter of `explicate_assign`. We'll need to use `cont` to recursively process both branches of the `if`, and we do not want to duplicate code, so we generate the following block using an auxiliary function named `create_block` that we discuss below.


```
(define (explicate_pred cnd thn els)
  (match cnd
    [(Var x) ___]
    [(Let x rhs body) ___]
    [(Prim 'not (list e)) ___]
    [(Prim op es) #:when (or (eq? op 'eq?) (eq? op '<))
      (IfStmt (Prim op es) (create_block thn)
               (create_block els)))]
    [(Bool b) (if b thn els)]
    [(If cnd^ thn^ els^) ___]
    [else (error "explicate_pred unhandled case" cnd)]))
```

Figure 4.13

Skeleton for the `explicate_pred` auxiliary function.

```
block_6:
  return (+ x 2)
```

We then use `goto block_6;` as the `cont` argument for compiling the branches. So the two branches compile to

```
      x = 40;                and      x = 777;
      goto block_6;          goto block_6;
```

Finally, we delegate to `explicate_pred`, passing the condition `(eq? y 0)` and the above code for the branches.

4.8.2 Create Block

We recommend implementing the `create_block` auxiliary function as follows, using a global variable `basic-blocks` to store a dictionary that maps labels to *tail* expressions. The main idea is that `create_block` generates a new label and then associates the given *tail* with the new label in the `basic-blocks` dictionary. The result of `create_block` is a `Goto` to the new label. However, if the given *tail* is already a `Goto`, then there is no need to generate a new label and entry in `basic-blocks`; we can simply return that `Goto`.

```
(define (create_block tail)
  (match tail
    [(Goto label) (Goto label)]
    [else
     (let ([label (gensym 'block)])
       (set! basic-blocks (cons (cons label tail) basic-blocks))
       (Goto label)))]))
```

4.8.3 Explicate Predicate

The skeleton for the `explicate_pred` function is given in Figure 4.13. It takes three parameters: 1) `cnd`, the condition expression of the `if`, 2) `thn`, the code generated by `explicate` for the “then” branch, and 3) `els`, the code generated by

explicate for the “else” branch. The `explicate_pred` function should match on `cnd` with a case for every kind of expression that can have type `Boolean`.

Consider the case for comparison operators. We translate the comparison to an `if` statement whose branches are `goto` statements created by applying `create_block` to the code generated for the `thn` and `els` branches. Let us illustrate this translation by returning to the program with an `if` expression in tail position, shown again below. We invoke `explicate_pred` on its condition (`eq? x 0`) .

```
(let ([x (read)])
  (if (eq? x 0) 42 777))
```

The two branches 42 and 777 were already compiled to `return` statements, from which we now create the following blocks.

```
block_1:
  return 42;
block_2:
  return 777;
```

After that, `explicate_pred` compiles the comparison (`eq? x 0`) to the following `if` statement.

```
if (eq? x 0)
  goto block_1;
else
  goto block_2;
```

Next consider the case for Boolean constants. We perform a kind of partial evaluation and output either the `thn` or `els` branch depending on whether the constant is `#t` or `#f`. Let us illustrate this with the following program.

```
(if #t 42 777)
```

Again, the two branches 42 and 777 were compiled to `return` statements, so `explicate_pred` compiles the constant `#t` to the code for the “then” branch.

```
return 42;
```

This case demonstrates that we sometimes discard the `thn` or `els` blocks that are input to `explicate_pred`.

The case for `if` expressions in `explicate_pred` is particularly illuminating because it deals with the challenges we discussed above regarding nested `if` expressions (Figure 4.12). The `thn`[^] and `els`[^] branches of the `if` inherit their context from the current one, that is, predicate context. So you should recursively apply `explicate_pred` to the `thn`[^] and `els`[^] branches. For both of those recursive calls, pass `thn` and `els` as the extra parameters. Thus, `thn` and `els` may get used twice, once inside each recursive call. As discussed above, to avoid duplicating code, we need to add them to the dictionary of basic blocks so that we can instead refer to them by name and execute them with a `goto`.

Figure 4.12 shows the output of the `remove_complex_operands` pass and then the `explicate_control` pass on the example program. We walk through the output program. Following the order of evaluation in the output of `remove_complex_operands`, we first have two calls to `(read)` and then the comparison `(< x 1)` in the predicate of the inner `if`. In the output of `explicate_control`, in the block labeled `start`, are two assignment statements followed by a `if` statement that branches to `block_4` or `block_5`. The blocks associated with those labels contain the translations of the code `(eq? x 0)` and `(eq? x 2)`, respectively. In particular, we start `block_4` with the comparison `(eq? x 0)` and then branch to `block_2` or `block_3`, which correspond to the two branches of the outer `if`, i.e., `(+ y 2)` and `(+ y 10)`. The story for `block_5` is similar to that of `block_4`.

4.8.4 Interactions between Explicate and Shrink

The way in which the `shrink` pass transforms logical operations such as `and` and `or` can impact the quality of code generated by `explicate_control`. For example, consider the following program.

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

The `and` operation should transform into something that the `explicate_pred` function can still analyze and descend through to reach the underlying `eq?` conditions. Ideally, your `explicate_control` pass should generate code similar to the following for the above program.

```
start:
  tmp1 = (read);
  if (eq? tmp1 0) goto block40;
  else goto block39;
block40:
  tmp2 = (read);
  if (eq? tmp2 1) goto block38;
  else goto block39;
block38:
  return 0;
block39:
  return 42;
```

Exercise 23 Implement the pass `explicate_control` by adding the cases for Boolean constants and `if` to the `explicate_tail` and `explicate_assign` functions. Implement the auxiliary function `explicate_pred` for predicate contexts. Create test cases that exercise all of the new cases in the code for this pass. Add the following entry to the list of `passes` in `run-tests.rkt` and then run this script to test your compiler.

```
(list "explicate_control" explicate_control interp-Cif type-check-Cif)
```

4.9 Select Instructions

The `select_instructions` pass translates C_{if} to $x86_{\text{if}}^{\text{Var}}$. Recall that we implement this pass using three auxiliary functions, one for each of the non-terminals *atm*, *stmt*, and *tail* in C_{if} (Figure 4.8). For *atm*, we have new cases for the Booleans. We take the usual approach of encoding them as integers.

$$\#t \Rightarrow 1 \qquad \#f \Rightarrow 0$$

For translating statements, we discuss some of the cases. The `not` operation can be implemented in terms of `xorq` as we discussed at the beginning of this section. Given an assignment, if the left-hand side variable is the same as the argument of `not`, then just the `xorq` instruction suffices.

$$var = (\text{not } var); \Rightarrow \text{xorq } \$1, var$$

Otherwise, a `movq` is needed to adapt to the update-in-place semantics of x86. In the following translation, let *arg* be the result of translating *atm* to x86.

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

Next consider the cases for equality comparisons. Translating this operation to x86 is slightly involved due to the unusual nature of the `cmpq` instruction that we discussed in Section 4.4. We recommend translating an assignment with an equality on the right-hand side into a sequence of three instructions.

$$var = ((eq? \text{atm}_1 \text{atm}_2)); \Rightarrow \begin{array}{l} \text{cmpq } arg_2, arg_1 \\ \text{sete } \%al \\ \text{movzbq } \%al, var \end{array}$$

The translations for the other comparison operators are similar to the above but use different condition codes for the `set` instruction.

Regarding the *tail* non-terminal, we have two new cases: `goto` and `if` statements. Both are straightforward to translate to x86. A `goto` statement becomes a jump instruction.

$$\text{goto } \ell; \Rightarrow \text{jmp } \ell$$

An `if` statement becomes a compare instruction followed by a conditional jump (for the “then” branch) and the fall-through is to a regular jump (for the “else” branch).

$$\begin{array}{l} \text{if } (eq? \text{atm}_1 \text{atm}_2) \\ \quad \text{goto } \ell_1; \\ \text{else} \\ \quad \text{goto } \ell_2; \end{array} \Rightarrow \begin{array}{l} \text{cmpq } arg_2, arg_1 \\ \text{je } \ell_1 \\ \text{jmp } \ell_2 \end{array}$$

Again, the translations for the other comparison operators are similar to the above but use different condition codes for the conditional jump instruction.

Exercise 24 Expand your `select_instructions` pass to handle the new features of the C_{if} language. Add the following entry to the list of `passes` in `run-tests.rkt`

```
(list "select_instructions" select_instructions interp-pseudo-x86-1)
```

Run the script to test your compiler on all the test programs.

4.10 Register Allocation

The changes required for compiling \mathcal{L}_{if} affect liveness analysis, building the interference graph, and assigning homes, but the graph coloring algorithm itself does not change.

4.10.1 Liveness Analysis

Recall that for \mathcal{L}_{var} we implemented liveness analysis for a single basic block (Section 3.2). With the addition of `if` expressions to \mathcal{L}_{if} , `explicate_control` produces many basic blocks.

The first question is: in what order should we process the basic blocks? Recall that to perform liveness analysis on a basic block we need to know the live-after set for the last instruction in the block. If a basic block has no successors (i.e. contains no jumps to other blocks), then it has an empty live-after set and we can immediately apply liveness analysis to it. If a basic block has some successors, then we need to complete liveness analysis on those blocks first. These ordering constraints are the reverse of a *topological order* on a graph representation of the program. In particular, the *control flow graph* (CFG) (Allen 1970) of a program has a node for each basic block and an edge for each jump from one block to another. It is straightforward to generate a CFG from the dictionary of basic blocks. One then transposes the CFG and applies the topological sort algorithm. We recommend using the `tsort` and `transpose` functions of the Racket `graph` package to accomplish this. As an aside, a topological ordering is only guaranteed to exist if the graph does not contain any cycles. This is the case for the control-flow graphs that we generate from \mathcal{L}_{if} programs. However, in Chapter 5 we add loops to create $\mathcal{L}_{\text{while}}$ and learn how to handle cycles in the control-flow graph.

You'll need to construct a directed graph to represent the control-flow graph. Do not use the `directed-graph` of the `graph` package because that only allows at most one edge between each pair of vertices, but a control-flow graph may have multiple edges between a pair of vertices. The `multigraph.rkt` file in the support code implements a graph representation that allows multiple edges between a pair of vertices.

The next question is how to analyze jump instructions. Recall that in Section 3.2 we maintain an alist named `label->live` that maps each label to the set of live locations at the beginning of its block. We use `label->live` to determine the live-before set for each (`Jump label`) instruction. Now that we have many basic blocks, `label->live` needs to be updated as we process the blocks. In particular, after performing liveness analysis on a block, we take the live-before set of its first instruction and associate that with the block's label in the `label->live` alist.

In $\text{x86}_{\text{if}}^{\text{var}}$ we also have the conditional jump (`JumpIf cc label`) to deal with. Liveness analysis for this instruction is particularly interesting because, during compilation,

we do not know which way a conditional jump will go. So we do not know whether to use the live-before set for the block associated with the *label* or the live-before set for the following instruction. However, there is no harm to the correctness of the generated code if we classify more locations as live than the ones that are truly live during one particular execution of the instruction. Thus, we can take the union of the live-before sets from the following instruction and from the mapping for *label* in *label*->live.

The auxiliary functions for computing the variables in an instruction's argument and for computing the variables read-from (*R*) or written-to (*W*) by an instruction need to be updated to handle the new kinds of arguments and instructions in $x86_{lf}^{Var}$.

Exercise 25 Update the `uncover_live` pass to apply liveness analysis to every basic block in the program. Add the following entry to the list of `passes` in the `run-tests.rkt` script.

```
(list "uncover_live" uncover_live interp-pseudo-x86-1)
```

4.10.2 Build the Interference Graph

Many of the new instructions in $x86_{lf}^{Var}$ can be handled in the same way as the instructions in $x86_{Var}$. Some instructions, e.g., the `movzbq` instruction, require special care, similar to the `movq` instruction. See rule number 1 in Section 3.3.

Exercise 26 Update the `build_interference` pass for $x86_{lf}^{Var}$. Add the following entries to the list of `passes` in the `run-tests.rkt` script.

```
(list "build_interference" build_interference interp-pseudo-x86-1)
(list "allocate_registers" allocate_registers interp-pseudo-x86-1)
```

4.11 Patch Instructions

The new instructions `cmpq` and `movzbq` have some special restrictions that need to be handled in the `patch_instructions` pass. The second argument of the `cmpq` instruction must not be an immediate value (such as an integer). So if you are comparing two immediates, we recommend inserting a `movq` instruction to put the second argument in `rax`. As usual, `cmpq` may have at most one memory reference. The second argument of the `movzbq` must be a register.

Exercise 27 Update `patch_instructions` pass for $x86_{lf}^{Var}$. Add the following entry to the list of `passes` in `run-tests.rkt` and then run this script to test your compiler.

```
(list "patch_instructions" patch_instructions interp-x86-1)
```

Figure 4.14 shows a simple example program in \mathcal{L}_{lf} translated to x86, showing the results of `explicate_control`, `select_instructions`, and the final x86 assembly.

Figure 4.15 lists all the passes needed for the compilation of \mathcal{L}_{lf} .

```

(if (eq? (read) 1) 42 0)
⇓
start:
  tmp7951 = (read);
  if (eq? tmp7951 1)
    goto block7952;
  else
    goto block7953;
block7952:
  return 42;
block7953:
  return 0;
⇓
start:
  callq read_int
  movq %rax, tmp7951
  cmpq $1, tmp7951
  je block7952
  jmp block7953
block7953:
  movq $0, %rax
  jmp conclusion
block7952:
  movq $42, %rax
  jmp conclusion

```

⇒

```

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

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

```

Figure 4.14

Example compilation of an if expression to x86, showing the results of `explicate_control`, `select_instructions`, and the final x86 assembly code.

4.12 Challenge: Optimize Blocks and Remove Jumps

We discuss two optional challenges that involve optimizing the control-flow of the program.

4.12.1 Optimize Blocks

The algorithm for `explicate_control` that we discussed in Section 4.8 sometimes generates too many blocks. It creates a basic block whenever a continuation *might* get used more than once (e.g., whenever the `cont` parameter is passed into two or more recursive calls). However, some continuation arguments may not be used at all. For example, consider the case for the constant `#t` in `explicate_pred`, where we discard the `els` continuation. The following example program falls into this case, and it creates two unused blocks.

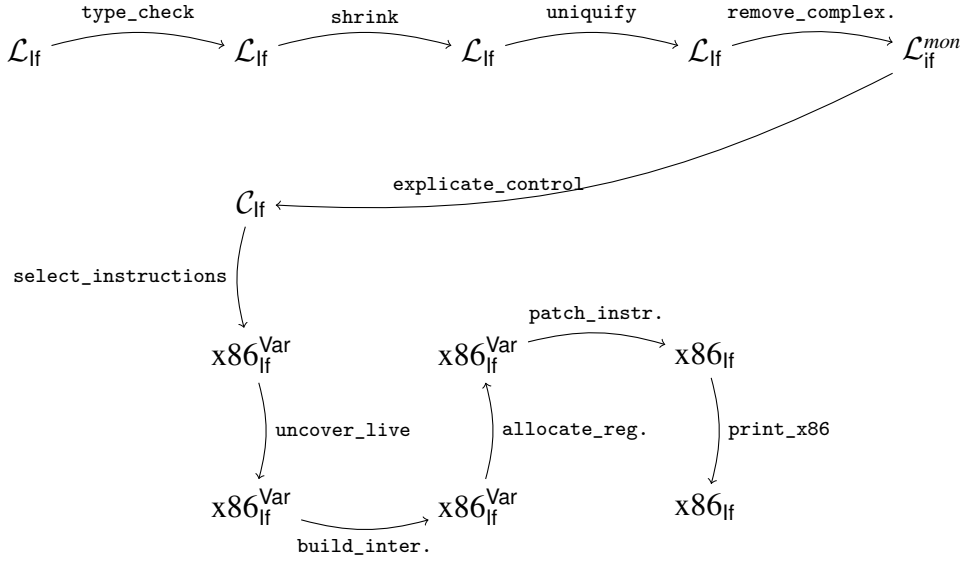


Figure 4.15

Diagram of the passes for \mathcal{L}_{if} , a language with conditionals.

```

(let ([y (if #t
  (read)
  (if (eq? (read) 0)
    777
    (let ([x (read)])
      (+ 1 x)))]])
  (+ y 2))
⇒
start:
  y = (read);
  goto block_5;
block_5:
  return (+ y 2);
block_6:
  y = 777;
  goto block_5;
block_7:
  x = (read);
  y = (+ 1 x2);
  goto block_5;
  
```

So the question is how can we decide whether to create a basic block? *Lazy evaluation* (Friedman and Wise 1976) can solve this conundrum by delaying the creation of a basic block until the point in time where we know it will be used. Racket provides support for lazy evaluation with the `racket/promise` package. The expression `(delay $e_1 \dots e_n$)` creates a *promise* in which the evaluation of the expressions is postponed. When `(force p)` is applied to a promise p for the first time, the expressions $e_1 \dots e_n$ are evaluated and the result of e_n is cached in the promise and returned. If `force` is applied again to the same promise, then the cached result is returned. If `force` is applied to an argument that is not a promise, `force` simply returns the argument.

We use promises for the input and output of the functions `explicate_pred`, `explicate_assign`, and `explicate_tail`. So instead of taking and returning


```

(let ([y (if #t
             (read)
             (if (eq? (read) 0)
                 777
                 (let ([x (read)])
                   (+ 1 x))))))]
      (+ y 2))
⇒
start:
  y = (read);
  goto block_5;
block_5:
  return (+ y 2);

```

Figure 4.16

Translation from \mathcal{L}_{lf} to \mathcal{C}_{lf} via the improved `explicate_control`.

tail expressions, they take and return promises. Furthermore, when we come to a situation in which a continuation might be used more than once, as in the case for `if` in `explicate_pred`, we create a delayed computation that creates a basic block for each continuation (if there is not already one) and then returns a `goto` statement to that basic block. When we come to a situation where we have a promise but need an actual piece of code, e.g. to create a larger piece of code with a constructor such as `Seq`, then insert a call to `force`. Also we must modify the `create_block` function to begin with `delay` to create a promise. When forced, this promise forces the original promise. If that returns a `Goto` (because the block was already added to `basic-blocks`), then we return the `Goto`. Otherwise we add the block to `basic-blocks` and return a `Goto` to the new label.

```

(define (create_block tail)
  (delay
    (define t (force tail))
    (match t
      [(Goto label) (Goto label)]
      [else
       (let ([label (gensym 'block)])
         (set! basic-blocks (cons (cons label tail) basic-blocks))
         (Goto label)))]))

```

Figure 4.16 shows the output of improved `explicate_control` on the above example. As you can see, the number of basic blocks has been reduced from 4 blocks (see Figure 4.12) down to 2 blocks.

Exercise 28 Implement the improvements to the `explicate_control` pass. Check that it removes trivial blocks in a few example programs. Then check that your compiler still passes all of your tests.

4.12.2 Remove Jumps

There is an opportunity for removing jumps that is apparent in the example of Figure 4.14. The `start` block ends with a jump to `block_5` and there are no other jumps to `block_5` in the rest of the program. In this situation we can avoid the runtime overhead of this jump by merging `block_5` into the preceding block, in

<pre> start: callq read_int movq %rax, %rcx jmp block_5 block_5: movq %rcx, %rax addq \$2, %rax jmp conclusion </pre>	\Rightarrow	<pre> start: callq read_int movq %rax, %rcx movq %rcx, %rax addq \$2, %rax jmp conclusion </pre>
---	---------------	--

Figure 4.17

Merging basic blocks by removing unnecessary jumps.

this case the `start` block. Figure 4.17 shows the output of `allocate_registers` on the left and the result of this optimization on the right.

Exercise 29 Implement a pass named `remove_jumps` that merges basic blocks into their preceding basic block, when there is only one preceding block. The pass should translate from $x86_{\text{if}}^{\text{Var}}$ to $x86_{\text{if}}^{\text{Var}}$. In the `run-tests.rkt` script, add the following entry to the list of passes between `allocate_registers` and `patch_instructions`.

```
(list "remove-jumps" remove-jumps interp-pseudo-x86-1)
```

Run the script to test your compiler. Check that `remove_jumps` accomplishes the goal of merging basic blocks on several test programs.

4.13 Further Reading

The algorithm for the `explicate_control` pass is based on the `expose-basic-blocks` pass in the course notes of Dybvig and Keep (2010). It has similarities to the algorithms of Danvy (2003) and Appel and Palsberg (2003), and is related to translations into continuation passing style (van Wijngaarden 1966; Fischer 1972; Reynolds 1972; Plotkin 1975; Friedman, Wand, and Haynes 2001). The treatment of conditionals in the `explicate_control` pass is similar to short-cut boolean evaluation (Logothetis and Mishra 1981; Aho et al. 2006; Clarke 1989; Danvy 2003) and the case-of-case transformation (Peyton Jones and Santos 1998).

5 Loops and Dataflow Analysis

In this chapter we study two features that are the hallmarks of imperative programming languages: loops and assignments to local variables. The following example demonstrates these new features by computing the sum of the first five positive integers.

```
(let ([sum 0])
  (let ([i 5])
    (begin
      (while (> i 0)
        (begin
          (set! sum (+ sum i))
          (set! i (- i 1))))
      sum)))
```

The `while` loop consists of a condition and a body⁹. The body is evaluated repeatedly so long as the condition remains true. The `set!` consists of a variable and a right-hand-side expression. The `set!` updates value of the variable to the value of the right-hand-side. The primary purpose of both the `while` loop and `set!` is to cause side effects, so they do not have a meaningful result value. Instead their result is the `#<void>` value. The expression `(void)` is an explicit way to create the `#<void>` value and it has type `Void`. The `#<void>` value can be passed around just like other values inside an $\mathcal{L}_{\text{While}}$ program and it can be compared for equality with another `#<void>` value. However, there are no other operations specific to the the `#<void>` value in $\mathcal{L}_{\text{While}}$. In contrast, Racket defines the `void?` predicate that returns `#t` when applied to `#<void>` and `#f` otherwise.¹⁰ With the addition of side-effecting features such as `while` loop and `set!`, it is helpful to also include in a language feature for sequencing side effects: the `begin` expression. It consists of one or more subexpressions that are evaluated left-to-right.

9. The `while` loop is not a built-in feature of the Racket language, but Racket includes many looping constructs and it is straightforward to define `while` as a macro.

10. Racket's `Void` type corresponds to what is often called the `Unit` type. Racket's `Void` type is inhabited by a single value `#<void>` which corresponds to `unit` or `()` in the literature (Pierce 2002).

<i>type</i>	::=	Integer
<i>exp</i>	::=	<i>int</i> (<i>read</i>) (<i>- exp</i>) (<i>+ exp exp</i>) (<i>- exp exp</i>)
<i>exp</i>	::=	<i>var</i> (<i>let</i> ([<i>var exp</i>]) <i>exp</i>)
<i>type</i>	::=	Boolean
<i>bool</i>	::=	#t #f
<i>cmp</i>	::=	eq? < <= > >=
<i>exp</i>	::=	<i>bool</i> (<i>and exp exp</i>) (<i>or exp exp</i>) (<i>not exp</i>)
		(<i>cmp exp exp</i>) (<i>if exp exp exp</i>)
<i>type</i>	::=	Void
<i>exp</i>	::=	(<i>set!</i> <i>var exp</i>) (<i>begin exp* exp</i>) (<i>while exp exp</i>) (<i>void</i>)
$\mathcal{L}_{\text{While}}$::=	<i>exp</i>

Figure 5.1

The concrete syntax of $\mathcal{L}_{\text{While}}$, extending \mathcal{L}_{If} (Figure 4.1).

<i>type</i>	::=	Integer
<i>op</i>	::=	read + -
<i>exp</i>	::=	(Int <i>int</i>) (Prim <i>op</i> (<i>exp</i> ...))
<i>exp</i>	::=	(Var <i>var</i>) (Let <i>var exp exp</i>)
<i>type</i>	::=	Boolean
<i>bool</i>	::=	#t #f
<i>cmp</i>	::=	eq? < <= > >=
<i>op</i>	::=	<i>cmp</i> and or not
<i>exp</i>	::=	(Bool <i>bool</i>) (If <i>exp exp exp</i>)
<i>type</i>	::=	Void
<i>exp</i>	::=	(SetBang <i>var exp</i>) (Begin <i>exp* exp</i>) (WhileLoop <i>exp exp</i>) (Void)
$\mathcal{L}_{\text{While}}$::=	(Program '() <i>exp</i>)

Figure 5.2

The abstract syntax of $\mathcal{L}_{\text{While}}$, extending \mathcal{L}_{If} (Figure 4.2).

5.1 The $\mathcal{L}_{\text{While}}$ Language

The concrete syntax of $\mathcal{L}_{\text{While}}$ is defined in Figure 5.1 and its abstract syntax is defined in Figure 5.2. The definitional interpreter for $\mathcal{L}_{\text{While}}$ is shown in Figure 5.3. We add new cases for **SetBang**, **WhileLoop**, **Begin**, and **Void** and we make changes to the cases for **Var** and **Let** regarding variables. To support assignment to variables and to make their lifetimes indefinite (see the second example in Section 8.2), we box the value that is bound to each variable (in **Let**). The case for **Var** unboxes the value. Now to discuss the new cases. For **SetBang**, we find the variable in the environment to obtain a boxed value and then we change it using **set-box!** to the result of evaluating the right-hand side. The result value of a **SetBang** is **#<void>**. For the **WhileLoop**, we repeatedly 1) evaluate the condition, and if the result is true, 2) evaluate the body. The result value of a **while** loop is also **#<void>**. The (**Begin** *es body*) expression evaluates the subexpressions *es* for their effects and

```

(define interp-Lwhile-class
  (class interp-Lif-class
    (super-new)

    (define/override ((interp-exp env) e)
      (define recur (interp-exp env))
      (match e
        [(Let x e body)
         (define new-env (dict-set env x (box (recur e))))
         ((interp-exp new-env) body)]
        [(Var x) (unbox (dict-ref env x))]
        [(SetBang x rhs)
         (set-box! (dict-ref env x) (recur rhs))]
        [(WhileLoop cnd body)
         (define (loop)
           (cond [(recur cnd) (recur body) (loop)]
                 [else      (void)]))
         (loop)]
        [(Begin es body)
         (for ([e es]) (recur e))
         (recur body)]
        [(Void) (void)]
        [else ((super interp-exp env) e)]))
  ))

(define (interp-Lwhile p)
  (send (new interp-Lwhile-class) interp-program p))

```

Figure 5.3Interpreter for $\mathcal{L}_{\text{While}}$.

then evaluates and returns the result from *body*. The (Void) expression produces the #<void> value.

The type checker for $\mathcal{L}_{\text{While}}$ is defined in Figure 5.4. The type checking of the SetBang expression requires the type of the variable and the right-hand-side to agree. The result type is Void. For while, the condition must be a Boolean and the result type is Void. For Begin, the result type is the type of its last subexpression.

At first glance, the translation of these language features to x86 seems straightforward because the \mathcal{C}_{If} intermediate language already supports all of the ingredients that we need: assignment, goto, conditional branching, and sequencing. However, there are complications that arise which we discuss in the next section. After that we introduce the changes necessary to the existing passes.

5.2 Cyclic Control Flow and Dataflow Analysis

Up until this point the programs generated in `explicate_control` were guaranteed to be acyclic. However, each **while** loop introduces a cycle. But does that matter? Indeed it does. Recall that for register allocation, the compiler performs liveness analysis to determine which variables can share the same register. To accomplish this we analyzed the control-flow graph in reverse topological order (Section 4.10.1), but topological order is only well-defined for acyclic graphs.

```

(define type-check-Lwhile-class
  (class type-check-Lif-class
    (super-new)
    (inherit check-type-equal?)

    (define/override (type-check-exp env)
      (lambda (e)
        (define recur (type-check-exp env))
        (match e
          [(SetBang x rhs)
           (define-values (rhs^ rhsT) (recur rhs))
           (define varT (dict-ref env x))
           (check-type-equal? rhsT varT e)
           (values (SetBang x rhs^) 'Void)]
          [(WhileLoop cnd body)
           (define-values (cnd^ Tc) (recur cnd))
           (check-type-equal? Tc 'Boolean e)
           (define-values (body^ Tbody) ((type-check-exp env) body))
           (values (WhileLoop cnd^ body^) 'Void)]
          [(Begin es body)
           (define-values (es^ ts)
             (for/lists (l1 l2) ([e es]) (recur e)))
           (define-values (body^ Tbody) (recur body))
           (values (Begin es^ body^) Tbody)]
          [else ((super type-check-exp env) e)])))
    ))

(define (type-check-Lwhile p)
  (send (new type-check-Lwhile-class) type-check-program p))

```

Figure 5.4

Type checker for the $\mathcal{L}_{\text{While}}$ language.

Let us return to the example of computing the sum of the first five positive integers. Here is the program after instruction selection but before register allocation.

```

(define (main) : Integer
  mainstart:
    movq $0, sum
    movq $5, i
    jmp block5
  block5:
    movq i, tmp3
    cmpq tmp3, $0
    jl block7
    jmp block8
                                block7:
                                addq i, sum
                                movq $1, tmp4
                                negq tmp4
                                addq tmp4, i
                                jmp block5
                                block8:
                                movq $27, %rax
                                addq sum, %rax
                                jmp mainconclusion
                                )

```

Recall that liveness analysis works backwards, starting at the end of each function. For this example we could start with `block8` because we know what is live at the beginning of the conclusion, just `rax` and `rsp`. So the live-before set for `block8` is

`{rsp, sum}`. Next we might try to analyze `block5` or `block7`, but `block5` jumps to `block7` and vice versa, so it seems that we are stuck.

The way out of this impasse is to realize that we can compute an under-approximation of each live-before set by starting with empty live-after sets. By *under-approximation*, we mean that the set only contains variables that are live for some execution of the program, but the set may be missing some variables that are live. Next, the under-approximations for each block can be improved by 1) updating the live-after set for each block using the approximate live-before sets from the other blocks and 2) perform liveness analysis again on each block. In fact, by iterating this process, the under-approximations eventually become the correct solutions! This approach of iteratively analyzing a control-flow graph is applicable to many static analysis problems and goes by the name *dataflow analysis*. It was invented by Kildall (1973) in his Ph.D. thesis at the University of Washington.

Let us apply this approach to the above example. We use the empty set for the initial live-before set for each block. Let m_0 be the following mapping from label names to sets of locations (variables and registers).

```
mainstart: {}, block5: {}, block7: {}, block8: {}
```

Using the above live-before approximations, we determine the live-after for each block and then apply liveness analysis to each block. This produces our next approximation m_1 of the live-before sets.

```
mainstart: {}, block5: {i}, block7: {i, sum}, block8: {rsp, sum}
```

For the second round, the live-after for `mainstart` is the current live-before for `block5`, which is `{i}`. So the liveness analysis for `mainstart` computes the empty set. The live-after for `block5` is the union of the live-before sets for `block7` and `block8`, which is `{i, rsp, sum}`. So the liveness analysis for `block5` computes `{i, rsp, sum}`. The live-after for `block7` is the live-before for `block5` (from the previous iteration), which is `{i}`. So the liveness analysis for `block7` remains `{i, sum}`. Together these yield the following approximation m_2 of the live-before sets.

```
mainstart: {}, block5: {i, rsp, sum}, block7: {i, sum}, block8: {rsp, sum}
```

In the preceding iteration, only `block5` changed, so we can limit our attention to `mainstart` and `block7`, the two blocks that jump to `block5`. As a result, the live-before sets for `mainstart` and `block7` are updated to include `rsp`, yielding the following approximation m_3 .

```
mainstart: {rsp}, block5: {i, rsp, sum}, block7: {i, rsp, sum}, block8: {rsp, sum}
```

Because `block7` changed, we analyze `block5` once more, but its live-before set remains `{i, rsp, sum}`. At this point our approximations have converged, so m_3 is the solution.

This iteration process is guaranteed to converge to a solution by the Kleene Fixed-Point Theorem, a general theorem about functions on lattices (Kleene 1952). Roughly speaking, a *lattice* is any collection that comes with a partial ordering \sqsubseteq on its elements, a least element \perp (pronounced bottom), and a join operator \sqcup .¹¹ When two elements are ordered $m_i \sqsubseteq m_j$, it means that m_j contains at least as much information as m_i , so we can think of m_j as a better-or-equal approximation than m_i . The bottom element \perp represents the complete lack of information, i.e., the worst approximation. The join operator takes two lattice elements and combines their information, i.e., it produces the least upper bound of the two.

A dataflow analysis typically involves two lattices: one lattice to represent abstract states and another lattice that aggregates the abstract states of all the blocks in the control-flow graph. For liveness analysis, an abstract state is a set of locations. We form the lattice L by taking its elements to be sets of locations, the ordering to be set inclusion (\subseteq), the bottom to be the empty set, and the join operator to be set union. We form a second lattice M by taking its elements to be mappings from the block labels to sets of locations (elements of L). We order the mappings point-wise, using the ordering of L . So given any two mappings m_i and m_j , $m_i \sqsubseteq_M m_j$ when $m_i(\ell) \subseteq m_j(\ell)$ for every block label ℓ in the program. The bottom element of M is the mapping \perp_M that sends every label to the empty set, i.e., $\perp_M(\ell) = \emptyset$.

We can think of one iteration of liveness analysis applied to the whole program as being a function f on the lattice M . It takes a mapping as input and computes a new mapping.

$$f(m_i) = m_{i+1}$$

Next let us think for a moment about what a final solution m_s should look like. If we perform liveness analysis using the solution m_s as input, we should get m_s again as the output. That is, the solution should be a *fixed point* of the function f .

$$f(m_s) = m_s$$

Furthermore, the solution should only include locations that are forced to be there by performing liveness analysis on the program, so the solution should be the *least* fixed point.

The Kleene Fixed-Point Theorem states that if a function f is monotone (better inputs produce better outputs), then the least fixed point of f is the least upper bound of the *ascending Kleene chain* obtained by starting at \perp and iterating f as follows.

$$\perp \sqsubseteq f(\perp) \sqsubseteq f(f(\perp)) \sqsubseteq \dots \sqsubseteq f^n(\perp) \sqsubseteq \dots$$

When a lattice contains only finitely-long ascending chains, then every Kleene chain tops out at some fixed point after some number of iterations of f .

$$\perp \sqsubseteq f(\perp) \sqsubseteq f(f(\perp)) \sqsubseteq \dots \sqsubseteq f^k(\perp) = f^{k+1}(\perp) = m_s$$

11. Technically speaking, we will be working with join semi-lattices.


```

(define (analyze_dataflow G transfer bottom join)
  (define mapping (make-hash))
  (for ([v (in-vertices G)])
    (dict-set! mapping v bottom))
  (define worklist (make-queue))
  (for ([v (in-vertices G)])
    (enqueue! worklist v))
  (define trans-G (transpose G))
  (while (not (queue-empty? worklist))
    (define node (dequeue! worklist))
    (define input (for/fold ([state bottom])
      ([pred (in-neighbors trans-G node)])
      (join state (dict-ref mapping pred))))
    (define output (transfer node input))
    (cond [(not (equal? output (dict-ref mapping node)))]
      (dict-set! mapping node output)
      (for ([v (in-neighbors G node)])
        (enqueue! worklist v))))
    mapping)

```

Figure 5.5

Generic work list algorithm for dataflow analysis

The liveness analysis is indeed a monotone function and the lattice M only has finitely-long ascending chains because there are only a finite number of variables and blocks in the program. Thus we are guaranteed that iteratively applying liveness analysis to all blocks in the program will eventually produce the least fixed point solution.

Next let us consider dataflow analysis in general and discuss the generic work list algorithm (Figure 5.5). The algorithm has four parameters: the control-flow graph G , a function **transfer** that applies the analysis to one block, the **bottom** and **join** operator for the lattice of abstract states. The **analyze_dataflow** function is formulated as a *forward* dataflow analysis, that is, the inputs to the transfer function come from the predecessor nodes in the control-flow graph. However, liveness analysis is a *backward* dataflow analysis, so in that case one must supply the **analyze_dataflow** function with the transpose of the control-flow graph.

The algorithm begins by creating the bottom mapping, represented by a hash table. It then pushes all of the nodes in the control-flow graph onto the work list (a queue). The algorithm repeats the **while** loop as long as there are items in the work list. In each iteration, a node is popped from the work list and processed. The **input** for the node is computed by taking the join of the abstract states of all the predecessor nodes. The **transfer** function is then applied to obtain the **output** abstract state. If the output differs from the previous state for this block, the mapping for this block is updated and its successor nodes are pushed onto the work list.

5.3 Mutable Variables & Remove Complex Operands

There is a subtle interaction between the `remove_complex_operands` pass, the addition of `set!`, and the left-to-right order of evaluation of Racket. Consider the following example.

```
(let ([x 2])
  (+ x (begin (set! x 40) x)))
```

The result of this program is 42 because the first read from `x` produces 2 and the second produces 40. However, if we naively apply the `remove_complex_operands` pass to this example we obtain the following program whose result is 80!

```
(let ([x 2])
  (let ([tmp (begin (set! x 40) x)])
    (+ x tmp)))
```

The problem is that, with mutable variables, the ordering between reads and writes is important, and the `remove_complex_operands` pass moved the `set!` to happen before the first read of `x`.

We recommend solving this problem by giving special treatment to reads from mutable variables, that is, variables that occur on the left-hand side of a `set!`. We mark each read from a mutable variable with the form `get!` (`GetBang` in abstract syntax) to indicate that the read operation is effectful in that it can produce different results at different points in time. Let's apply this idea to the following variation that also involves a variable that is not mutated.

```
(let ([x 2])
  (let ([y 0])
    (+ y (+ x (begin (set! x 40) x))))))
```

We first analyze the above program to discover that variable `x` is mutable but `y` is not. We then transform the program as follows, replacing each occurrence of `x` with `(get! x)`.

```
(let ([x 2])
  (let ([y 0])
    (+ y (+ (get! x) (begin (set! x 40) (get! x))))))
```

Now that we have a clear distinction between reads from mutable and immutable variables, we can apply the `remove_complex_operands` pass, where reads from immutable variables are still classified as atomic expressions but reads from mutable variables are classified as complex. Thus, `remove_complex_operands` yields the following program.

```
(let ([x 2])
  (let ([y 0])
    (+ y (let ([t1 (get! x)])
            (let ([t2 (begin (set! x 40) (get! x))])
              (+ t1 t2)))))))
```

The temporary variable `t1` gets the value of `x` before the `set!`, so it is 2. The

temporary variable `t2` gets the value of `x` after the `set!`, so it is 40. We do not generate a temporary variable for the occurrence of `y` because it's an immutable variable. We want to avoid such unnecessary extra temporaries because they would needlessly increase the number of variables, making it more likely for some of them to be spilled. The result of this program is 42, the same as the result prior to `remove_complex_operands`.

The approach that we've sketched above requires only a small modification to `remove_complex_operands` to handle `get!`. However, it requires a new pass, called `uncover-get!`, that we discuss in Section 5.4.

As an aside, this problematic interaction between `set!` and the pass `remove_complex_operands` is particular to Racket and not its predecessor, the Scheme language. The key difference is that Scheme does not specify an order of evaluation for the arguments of an operator or function call (Sperber et al. 2009). Thus, a compiler for Scheme is free to choose any ordering: both 42 and 80 would be correct results for the example program. Interestingly, Racket is implemented on top of the Chez Scheme compiler (Dybvig 2006) and an approach similar to the one presented in this section (using extra `let` bindings to control the order of evaluation) is used in the translation from Racket to Scheme (Flatt et al. 2019).

Having discussed the complications that arise from adding support for assignment and loops, we turn to discussing the individual compilation passes.

5.4 Uncover get!

The goal of this pass is to mark uses of mutable variables so that `remove_complex_operands` can treat them as complex expressions and thereby preserve their ordering relative to the side-effects in other operands. So the first step is to collect all the mutable variables. We recommend creating an auxiliary function for this, named `collect-set!`, that recursively traverses expressions, returning the set of all variables that occur on the left-hand side of a `set!`. Here's an excerpt of its implementation.

```
(define (collect-set! e)
  (match e
    [(Var x) (set)]
    [(Int n) (set)]
    [(Let x rhs body)
     (set-union (collect-set! rhs) (collect-set! body))]
    [(SetBang var rhs)
     (set-union (set var) (collect-set! rhs))]
    ...))
```

By placing this pass after `uniquify`, we need not worry about variable shadowing and our logic for `Let` can remain simple, as in the excerpt above.

The second step is to mark the occurrences of the mutable variables with the new `GetBang` AST node (`get!` in concrete syntax). The following is an excerpt of the `uncover-get!-exp` function, which takes two parameters: the set of mutable

atm	$::=$	$(Int\ int) \mid (Var\ var)$
exp	$::=$	$atm \mid (Prim\ 'read\ ())$
		$\mid (Prim\ '-\ (atm)) \mid (Prim\ '+\ (atm\ atm)) \mid (Prim\ '-\ (atm\ atm))$
		$\mid (Let\ var\ exp\ exp)$
<hr/>		
atm	$::=$	$(Bool\ bool)$
exp	$::=$	$(Prim\ not\ (atm)) \mid (Prim\ cmp\ (atm\ atm)) \mid (If\ exp\ exp\ exp)$
<hr/>		
atm	$::=$	$(Void)$
exp	$::=$	$(GetBang\ var) \mid (SetBang\ var\ exp) \mid (Begin\ (exp\ \dots)\ exp)$
		$\mid (WhileLoop\ exp\ exp)$
$\mathcal{L}_{While}^{mon}$	$::=$	$(Program\ '()\ exp)$

Figure 5.6

$\mathcal{L}_{While}^{mon}$ is \mathcal{L}_{While} in monadic normal form.

variables `set!-vars`, and the expression `e` to be processed. The case for `(Var x)` replaces it with `(GetBang x)` if it is a mutable variable or leaves it alone if not.

```
(define ((uncover-get!-exp set!-vars) e)
  (match e
    [(Var x)
     (if (set-member? set!-vars x)
         (GetBang x)
         (Var x))]
    ...))
```

To wrap things up, define the `uncover-get!` function for processing a whole program, using `collect-set!` to obtain the set of mutable variables and then `uncover-get!-exp` to replace their occurrences with `GetBang`.

5.5 Remove Complex Operands

The new language forms, `get!`, `set!`, `begin`, and `while` are all complex expressions. The subexpressions of `set!`, `begin`, and `while` are allowed to be complex. Figure 5.6 defines the output language $\mathcal{L}_{While}^{mon}$ of this pass.

As usual, when a complex expression appears in a grammar position that needs to be atomic, such as the argument of a primitive operator, we must introduce a temporary variable and bind it to the complex expression. This approach applies, unchanged, to handle the new language forms. For example, in the following code there are two `begin` expressions appearing as arguments to the `+` operator. The output of `rco_exp` is shown below, in which the `begin` expressions have been bound to temporary variables. Recall that `let` expressions in $\mathcal{L}_{While}^{mon}$ are allowed to have arbitrary expressions in their right-hand-side expression, so it is fine to place `begin` there.

<i>atm</i>	<i>::=</i>	<i>(Int int) (Var var)</i>
<i>exp</i>	<i>::=</i>	<i>atm (Prim 'read ()) (Prim '- (atm))</i>
		<i> (Prim '+ (atm atm)) (Prim '- (atm atm))</i>
<i>stmt</i>	<i>::=</i>	<i>(Assign (Var var) exp)</i>
<i>tail</i>	<i>::=</i>	<i>(Return exp) (Seq stmt tail)</i>
<hr/>		
<i>atm</i>	<i>::=</i>	<i>(Bool bool)</i>
<i>cmp</i>	<i>::=</i>	<i>eq? < <= > >=</i>
<i>exp</i>	<i>::=</i>	<i>(Prim 'not (atm)) (Prim 'cmp (atm atm))</i>
<i>tail</i>	<i>::=</i>	<i>(Goto label)</i>
		<i> (IfStmt (Prim cmp (atm atm)) (Goto label) (Goto label))</i>
<hr/>		
<i>atm</i>	<i>::=</i>	<i>(Void)</i>
<i>stmt</i>	<i>::=</i>	<i>(Prim 'read ())</i>
\mathcal{C}_\circ	<i>::=</i>	<i>(CProgram info ((label . tail) ...))</i>

Figure 5.7

The abstract syntax of \mathcal{C}_\circ , extending \mathcal{C}_ff (Figure 4.8).

<pre>(let ([x2 10]) (let ([y3 0]) (+ (+ (begin (set! y3 (read)) (get! x2)) (begin (set! x2 (read)) (get! y3))) (get! x2))))</pre>	\Rightarrow	<pre>(let ([x2 10]) (let ([y3 0]) (let ([tmp4 (begin (set! y3 (read)) x2)]) (let ([tmp5 (begin (set! x2 (read)) y3)]) (let ([tmp6 (+ tmp4 tmp5)]) (let ([tmp7 x2]) (+ tmp6 tmp7))))))))</pre>
---	---------------	---

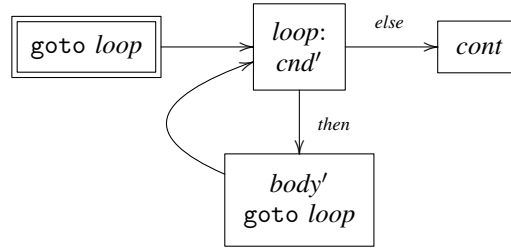
5.6 Explicate Control and \mathcal{C}_\circ

Recall that in the `explicate_control` pass we define one helper function for each kind of position in the program. For the \mathcal{L}_var language of integers and variables we needed assignment and tail positions. The `if` expressions of \mathcal{L}_ff introduced predicate positions. For \mathcal{L}_while , the `begin` expression introduces yet another kind of position: effect position. Except for the last subexpression, the subexpressions inside a `begin` are evaluated only for their effect. Their result values are discarded. We can generate better code by taking this fact into account.

The output language of `explicate_control` is \mathcal{C}_\circ (Figure 5.7), which is nearly identical to \mathcal{C}_ff . The only syntactic difference is the addition of `(Void)` and that `read` may appear as a statement. The most significant difference between the programs generated by `explicate_control` in Chapter 4 versus `explicate_control` in this chapter is that the control-flow graphs of the later may contain cycles.

The new auxiliary function `explicate_effect` takes an expression (in an effect position) and the code for its continuation. The function returns a *tail* that

includes the generated code for the input expression followed by the continuation. If the expression is obviously pure, that is, never causes side effects, then the expression can be removed, so the result is just the continuation. The case for (`WhileLoop cnd body`) expressions is interesting; the generated code is depicted in the following diagram.



We start by creating a fresh label `loop` for the top of the loop. Next, recursively process the `body` (in effect position) with a `goto` to `loop` as the continuation, producing `body'`. Process the `cnd` (in predicate position) with `body'` as the then-branch and the continuation block as the else-branch. The result should be added to the dictionary of `basic-blocks` with the label `loop`. The result for the whole `while` loop is a `goto` to the `loop` label.

The auxiliary functions for tail, assignment, and predicate positions need to be updated. The three new language forms, `while`, `set!`, and `begin`, can appear in assignment and tail positions. Only `begin` may appear in predicate positions; the other two have result type `Void`.

5.7 Select Instructions

Only two small additions are needed in the `select_instructions` pass to handle the changes to \mathcal{C}_\cup . First, to handle the addition of `(Void)` we simply translate it to `0`. Second, `read` may appear as a stand-alone statement instead of only appearing on the right-hand side of an assignment statement. The code generation is nearly identical to the one for assignment; just leave off the instruction for moving the result into the left-hand side.

5.8 Register Allocation

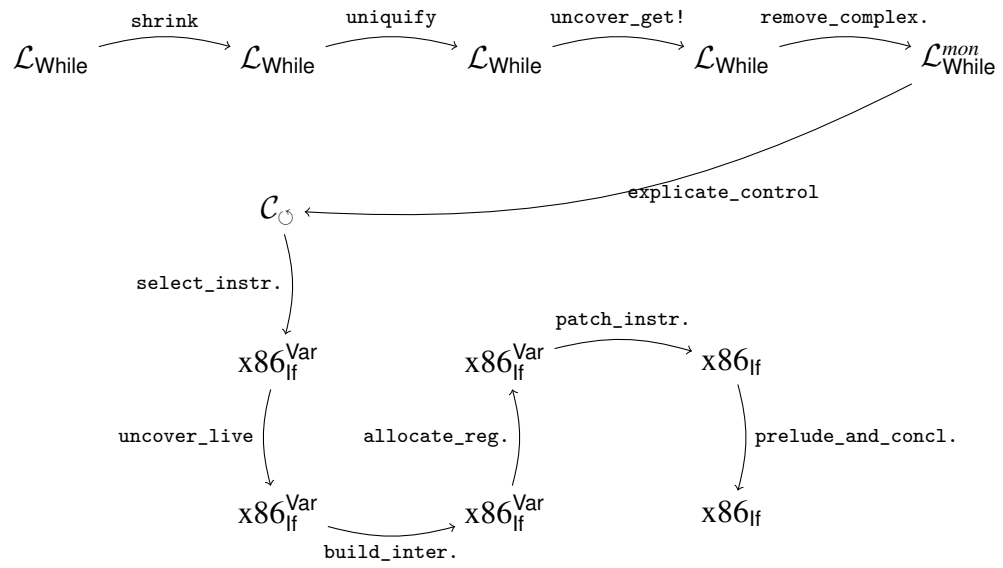
As discussed in Section 5.2, the presence of loops in $\mathcal{L}_{\text{While}}$ means that the control-flow graphs may contain cycles, which complicates the liveness analysis needed for register allocation. We recommend using the generic `analyze_dataflow` function that was presented at the end of Section 5.2 to perform liveness analysis, replacing the code in `uncover_live` that processed the basic blocks in topological order (Section 4.10.1).

The `analyze_dataflow` function has four parameters.

1. The first parameter `G` should be passed the transpose of the control-flow graph.

2. The second parameter **transfer** should be passed a function that applies liveness analysis to a basic block. It takes two parameters: the label for the block to analyze and the live-after set for that block. The transfer function should return the live-before set for the block. Also, as a side-effect, it should update the block's *info* with the liveness information for each instruction. To implement the **transfer** function, you should be able to reuse the code you already have for analyzing basic blocks.
3. The third and fourth parameters of **analyze_dataflow** are **bottom** and **join** for the lattice of abstract states, i.e. sets of locations. For liveness analysis, the bottom of the lattice is the empty set and the join operator is set union.

Figure 5.8 provides an overview of all the passes needed for the compilation of $\mathcal{L}_{\text{While}}$.

**Figure 5.8**Diagram of the passes for $\mathcal{L}_{\text{While}}$.

6 Tuples and Garbage Collection

In this chapter we study the implementation of tuples, called vectors in Racket. A tuple is a fixed-length sequence of elements where each element may have a different type. This language feature is the first to use the computer’s *heap* because the lifetime of a tuple is indefinite, that is, a tuple lives forever from the programmer’s viewpoint. Of course, from an implementer’s viewpoint, it is important to reclaim the space associated with a tuple when it is no longer needed, which is why we also study *garbage collection* techniques in this chapter.

Section 6.1 introduces the $\mathcal{L}_{\text{Tuple}}$ language including its interpreter and type checker. The $\mathcal{L}_{\text{Tuple}}$ language extends the $\mathcal{L}_{\text{While}}$ language of Chapter 5 with tuples. Section 6.2 describes a garbage collection algorithm based on copying live tuples back and forth between two halves of the heap. The garbage collector requires coordination with the compiler so that it can find all of the live tuples. Sections 6.4 through 6.9 discuss the necessary changes and additions to the compiler passes, including a new compiler pass named `expose_allocation`.

6.1 The $\mathcal{L}_{\text{Tuple}}$ Language

Figure 6.1 defines the concrete syntax for $\mathcal{L}_{\text{Tuple}}$ and Figure 6.2 defines the abstract syntax. The $\mathcal{L}_{\text{Tuple}}$ language includes the forms: `vector` for creating a tuple, `vector-ref` for reading an element of a tuple, `vector-set!` for writing to an element of a tuple, and `vector-length` for obtaining the number of elements of a tuple. The program below shows an example use of tuples. It creates a tuple `t` containing the elements 40, `#t`, and another tuple that contains just 2. The element at index 1 of `t` is `#t`, so the “then” branch of the `if` is taken. The element at index 0 of `t` is 40, to which we add 2, the element at index 0 of the tuple. So the result of the program is 42.

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

Tuples raise several interesting new issues. First, variable binding performs a shallow-copy when dealing with tuples, which means that different variables can

<i>type</i>	::=	Integer
<i>exp</i>	::=	<i>int</i> (<i>read</i>) (<i>- exp</i>) (<i>+ exp exp</i>) (<i>- exp exp</i>)
<i>exp</i>	::=	<i>var</i> (<i>let</i> ([<i>var exp</i>]) <i>exp</i>)
<i>type</i>	::=	Boolean
<i>bool</i>	::=	#t #f
<i>cmp</i>	::=	eq? < <= > >=
<i>exp</i>	::=	<i>bool</i> (<i>and exp exp</i>) (<i>or exp exp</i>) (<i>not exp</i>) (<i>cmp exp exp</i>) (<i>if exp exp exp</i>)
<i>type</i>	::=	Void
<i>exp</i>	::=	(<i>set!</i> <i>var exp</i>) (<i>begin exp* exp</i>) (<i>while exp exp</i>) (<i>void</i>)
<i>type</i>	::=	(Vector <i>type*</i>)
<i>exp</i>	::=	(vector <i>exp*</i>) (vector-length <i>exp</i>) (vector-ref <i>exp int</i>) (vector-set! <i>exp int exp</i>)
\mathcal{L}_{Tup}	::=	<i>exp</i>

Figure 6.1

The concrete syntax of \mathcal{L}_{Tup} , extending $\mathcal{L}_{\text{While}}$ (Figure 5.1).

<i>type</i>	::=	Integer
<i>op</i>	::=	<i>read</i> + -
<i>exp</i>	::=	(<i>Int int</i>) (<i>Prim op (exp ...)</i>)
<i>exp</i>	::=	(<i>Var var</i>) (<i>Let var exp exp</i>)
<i>type</i>	::=	Boolean
<i>bool</i>	::=	#t #f
<i>cmp</i>	::=	eq? < <= > >=
<i>op</i>	::=	<i>cmp</i> <i>and</i> <i>or</i> <i>not</i>
<i>exp</i>	::=	(<i>Bool bool</i>) (<i>If exp exp exp</i>)
<i>type</i>	::=	Void
<i>exp</i>	::=	(<i>SetBang var exp</i>) (<i>Begin exp* exp</i>) (<i>WhileLoop exp exp</i>) (<i>Void</i>)
<i>type</i>	::=	(Vector <i>type*</i>)
<i>op</i>	::=	vector vector-length
<i>exp</i>	::=	(Prim vector-ref (<i>exp</i> (<i>Int int</i>))) (Prim vector-set! (<i>exp</i> (<i>Int int</i>) <i>exp</i>)) (HasType <i>exp type</i>)
\mathcal{L}_{Tup}	::=	(Program '() <i>exp</i>)

Figure 6.2

The abstract syntax of \mathcal{L}_{Tup} .

refer to the same tuple, that is, two variables can be *aliases* for the same entity. Consider the following example in which both **t1** and **t2** refer to the same tuple value but **t3** refers to a different tuple value but with equal elements. The result of the program is 42.

```
(let ([t1 (vector 3 7)])
  (let ([t2 t1])
    (let ([t3 (vector 3 7)])
      (if (and (eq? t1 t2) (not (eq? t1 t3)))
          42
          0))))
```

Whether two variables are aliased or not affects what happens when the underlying tuple is mutated. Consider the following example in which `t1` and `t2` again refer to the same tuple value.

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

The mutation through `t2` is visible when referencing the tuple from `t1`, so the result of this program is 42.

The next issue concerns the lifetime of tuples. When does their lifetime end? Notice that $\mathcal{L}_{\text{Tuple}}$ does not include an operation for deleting tuples. Furthermore, the lifetime of a tuple is not tied to any notion of static scoping. For example, the following program returns 42 even though the variable `w` goes out of scope prior to the `vector-ref` that reads from the vector it was bound to.

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

From the perspective of programmer-observable behavior, tuples live forever. However, if they really lived forever then many long-running programs would run out of memory. To solve this problem, the language's runtime system performs automatic garbage collection.

Figure 6.3 shows the definitional interpreter for the $\mathcal{L}_{\text{Tuple}}$ language. We define the `vector`, `vector-ref`, `vector-set!`, and `vector-length` operations for $\mathcal{L}_{\text{Tuple}}$ in terms of the corresponding operations in Racket. One subtle point is that the `vector-set!` operation returns the `#<void>` value.

Figure 6.4 shows the type checker for $\mathcal{L}_{\text{Tuple}}$, which deserves some explanation. When allocating a tuple, we need to know which elements of the tuple are themselves tuples for the purposes of garbage collection. We can obtain this information during type checking. The type checker in Figure 6.4 not only computes the type of an expression, it also wraps every tuple creation with the form `(HasType e T)`, where `T` is the tuple's type. To create the s-expression for the `Vector` type in Figure 6.4, we use the unquote-splicing operator `,@` to insert the list `t*` without its usual start and end parentheses.

```

(define interp-Lvec-class
  (class interp-Lwhile-class
    (super-new)

    (define/override (interp-op op)
      (match op
        ['eq? (lambda (v1 v2)
                  (cond [(or (and (fixnum? v1) (fixnum? v2))
                             (and (boolean? v1) (boolean? v2))
                             (and (vector? v1) (vector? v2))
                             (and (void? v1) (void? v2)))
                        (eq? v1 v2)]))]
        ['vector vector]
        ['vector-length vector-length]
        ['vector-ref vector-ref]
        ['vector-set! vector-set!]
        [else (super interp-op op)]
      ))

    (define/override ((interp-exp env) e)
      (match e
        [(HasType e t) ((interp-exp env) e)]
        [else ((super interp-exp env) e)]
      ))
  ))

(define (interp-Lvec p)
  (send (new interp-Lvec-class) interp-program p))

```

Figure 6.3

Interpreter for the \mathcal{L}_{Tup} language.

6.2 Garbage Collection

Garbage collection is a runtime technique for reclaiming space on the heap that will not be used in the future of the running program. We use the term *object* to refer to any value that is stored in the heap, which for now only includes tuples.¹² Unfortunately, it is impossible to know precisely which objects will be accessed in the future and which will not. Instead, garbage collectors over approximate the set of objects that will be accessed by identifying which objects can possibly be accessed. The running program can directly access objects that are in registers and on the procedure call stack. It can also transitively access the elements of tuples, starting with a tuple whose address is in a register or on the procedure call stack. We define the *root set* to be all the tuple addresses that are in registers or on the procedure call stack. We define the *live objects* to be the objects that are reachable from the root set. Garbage collectors reclaim the space that is allocated to objects

12. The term “object” as it is used in the context of object-oriented programming has a more specific meaning than how we are using the term here.

```

(define type-check-Lvec-class
  (class type-check-Lif-class
    (super-new)
    (inherit check-type-equal?)

    (define/override (type-check-exp env)
      (lambda (e)
        (define recur (type-check-exp env))
        (match e
          [(Prim 'vector es)
            (define-values (e* t*) (for/lists (e* t*) ([e es]) (recur e)))
            (define t `(Vector ,@t*))
            (values (HasType (Prim 'vector e*) t) t)]
          [(Prim 'vector-ref (list e1 (Int i)))
            (define-values (e1^ t) (recur e1))
            (match t
              [(Vector ,ts ...)
                (unless (and (0 . <= . i) (i . < . (length ts)))
                  (error 'type-check "index -a out of bounds\nin ~v" i e))
                (values (Prim 'vector-ref (list e1^ (Int i))) (list-ref ts i))]
              [else (error 'type-check "expect Vector, not -a\nin ~v" t e)]))]
          [(Prim 'vector-set! (list e1 (Int i) arg) )
            (define-values (e-vec t-vec) (recur e1))
            (define-values (e-arg^ t-arg) (recur arg))
            (match t-vec
              [(Vector ,ts ...)
                (unless (and (0 . <= . i) (i . < . (length ts)))
                  (error 'type-check "index -a out of bounds\nin ~v" i e))
                (check-type-equal? (list-ref ts i) t-arg e)
                (values (Prim 'vector-set! (list e-vec (Int i) e-arg^)) 'Void)]
              [else (error 'type-check "expect Vector, not -a\nin ~v" t-vec e)]))]
          [(Prim 'vector-length (list e))
            (define-values (e^ t) (recur e))
            (match t
              [(Vector ,ts ...)
                (values (Prim 'vector-length (list e^)) 'Integer)]
              [else (error 'type-check "expect Vector, not -a\nin ~v" t e)]))]
          [(Prim 'eq? (list arg1 arg2))
            (define-values (e1 t1) (recur arg1))
            (define-values (e2 t2) (recur arg2))
            (match* (t1 t2)
              [(Vector ,ts1 ...) (Vector ,ts2 ...) (void)]
              [(other wise) (check-type-equal? t1 t2 e)])
            (values (Prim 'eq? (list e1 e2)) 'Boolean)]
          [(HasType (Prim 'vector es) t)
            ((type-check-exp env) (Prim 'vector es))]
          [(HasType e1 t)
            (define-values (e1^ t^) (recur e1))
            (check-type-equal? t t^ e)
            (values (HasType e1^ t) t)]
          [else ((super type-check-exp env) e)]
        )))
  ))

(define (type-check-Lvec p)
  (send (new type-check-Lvec-class) type-check-program p))

```

Figure 6.4

Type checker for the \mathcal{L}_{Tup} language.

that are no longer live. That means that some objects may not get reclaimed as soon as they could be, but at least garbage collectors do not reclaim the space dedicated to objects that will be accessed in the future! The programmer can influence which objects get reclaimed by causing them to become unreachable.

So the goal of the garbage collector is twofold:

1. preserve all the live objects, and
2. reclaim the memory of everything else, that is, the *garbage*.

6.2.1 Two-Space Copying Collector

Here we study a relatively simple algorithm for garbage collection that is the basis of many state-of-the-art garbage collectors (Lieberman and Hewitt 1983; Ungar 1984; Jones and Lins 1996; Detlefs et al. 2004; Dybvig 2006; Tene, Iyengar, and Wolf 2011). In particular, we describe a two-space copying collector (Wilson 1992) that uses Cheney’s algorithm to perform the copy (Cheney 1970). Figure 6.5 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 named the FromSpace and the ToSpace. Initially, all allocations go to the FromSpace until there is not enough room for the next allocation request. At that point, the garbage collector goes to work to make room for the next allocation.

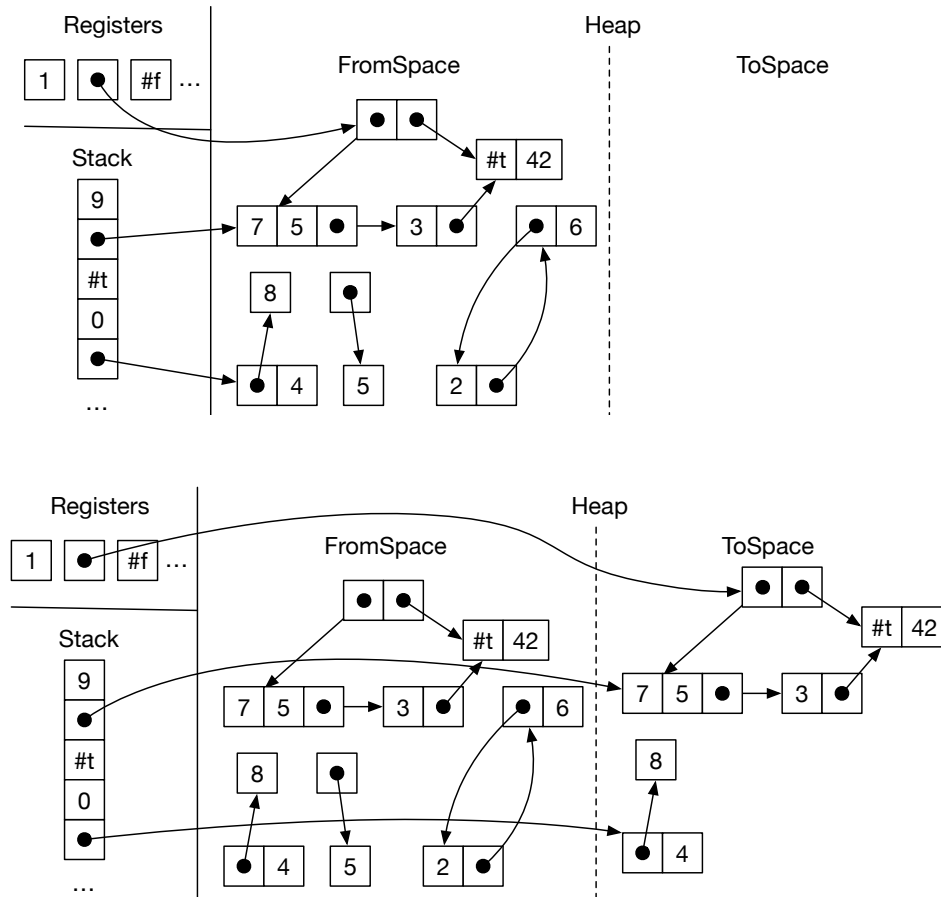
A copying collector makes more room by copying all of the live objects from the FromSpace into the ToSpace and then performs a sleight of hand, treating the ToSpace as the new FromSpace and the old FromSpace as the new ToSpace. In the example of Figure 6.5, the root set consists of three pointers, one in a register and two on the stack. All of the live objects have been copied to the ToSpace (the right-hand side of Figure 6.5) in a way that preserves the pointer relationships. For example, the pointer in the register still points to a tuple that in turn points to two other tuples. There are four tuples that are not reachable from the root set and therefore do not get copied into the ToSpace.

The exact situation in Figure 6.5 cannot be created by a well-typed program in \mathcal{L}_{Top} because it contains a cycle. However, creating cycles will be possible once we get to \mathcal{L}_{Dyn} (Chapter 9). We design the garbage collector to deal with cycles to begin with so we will not need to revisit this issue.

6.2.2 Graph Copying via Cheney’s Algorithm

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

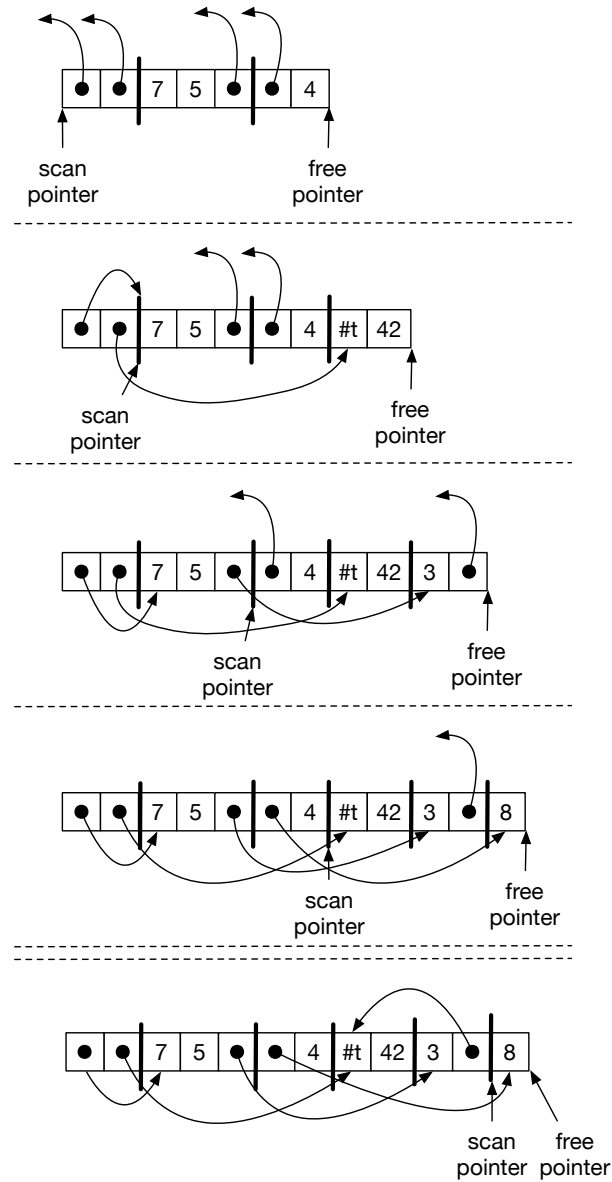
Figure 6.6 shows several snapshots of the ToSpace as the copy progresses. The queue is represented by a chunk of contiguous memory at the beginning of the ToSpace, using two pointers to track the front and the back of the queue, called the *free pointer* and the *scan pointer* respectively. The algorithm starts by copying all

**Figure 6.5**

A copying collector in action.

tuples that are immediately reachable from the root set into the ToSpace to form the initial queue. When we copy a tuple, we mark the old tuple to indicate that it has been visited. We discuss how this marking is accomplished in Section 6.2.3. 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 objects 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 objects.

Getting back to Figure 6.6, 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*

**Figure 6.6**

Depiction of the Cheney algorithm copying the live tuples.

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 queue is empty, that is, when the scan pointer catches up with the free pointer.

6.2.3 Data Representation

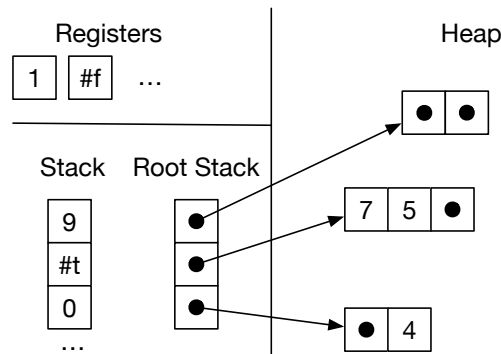
The garbage collector places some requirements on the data representations used by our compiler. First, the garbage collector needs to distinguish between pointers and other kinds of data such as integers. There are several ways to accomplish this.

1. Attached a tag to each object that identifies what type of object it is (McCarthy 1960).
2. Store different types of objects in different regions (Steele 1977).
3. Use type information from the program to either (a) generate type-specific code for collecting or (b) generate tables that guide the collector (Appel 1989; Goldberg 1991; Diwan, Moss, and Hudson 1992).

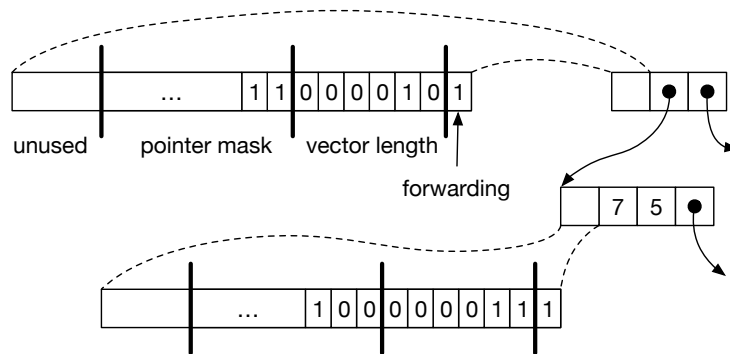
Dynamically typed languages, such as Racket, need to tag objects anyways, so option 1 is a natural choice for those languages. However, \mathcal{L}_{Tip} is a statically typed language, so it would be unfortunate to require tags on every object, especially small and pervasive objects like integers and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

Regarding the stack, we recommend using a separate stack for pointers, which we call the *root stack* (a.k.a. “shadow stack”) (Siebert 2001; Henderson 2002; Baker et al. 2009). That is, when a local variable needs to be spilled and is of type **Vector**, then we put it on the root stack instead of putting it on the procedure call stack. Furthermore, we always spill tuple-typed variables if they are live during a call to the collector, thereby ensuring that no pointers are in registers during a collection. Figure 6.7 reproduces the example from Figure 6.5 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 on the heap. We solve this problem by attaching a tag, an extra 64-bits, to each tuple. Figure 6.8 zooms in on the tags for two of the tuples in the example from Figure 6.5. 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 direction of the x86 shifting instructions **salq** (shift left) and **sarq** (shift right). Part of each tag is dedicated to specifying which elements of the tuple are pointers, the part labeled “pointer mask”. Within the pointer mask, a 1 bit indicates there is a pointer and a 0 bit indicates some other kind of data. The pointer mask starts at bit location 7. We limit tuples to a maximum size of

**Figure 6.7**

Maintaining a root stack to facilitate garbage collection.

**Figure 6.8**

Representation of tuples in the heap.

50 elements, so we just need 50 bits for the pointer mask.¹³ 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 a forwarding pointer. (The lower 3 bits of a pointer are always zero anyways because our tuples are 8-byte aligned.)

6.2.4 Implementation of the Garbage Collector

An implementation of the copying collector is provided in the `runtime.c` file. Figure 6.9 defines the interface to the garbage collector that is used by the compiler. The `initialize` function creates the FromSpace, ToSpace, and root stack and should be called in the prelude of the `main` function. The arguments of `initialize`

13. A production-quality compiler would handle arbitrary-sized tuples and use a more complex approach.

```

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 6.9

The compiler's interface to the garbage collector.

are the root stack size and the heap size. Both need to be multiples of 64 and 16384 is a good choice for both. The `initialize` function puts the address of the beginning of the FromSpace into the global variable `free_ptr`. The global variable `fromspace_end` points to the address that is 1-past the last element of the FromSpace. We use half-open intervals to represent chunks of memory (Dijkstra 1982). The `rootstack_begin` variable points to the first element of the root stack.

As long as there is room left in the FromSpace, your generated code can allocate tuples simply by moving the `free_ptr` forward. The amount of room left in the 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 there is enough room for the next allocation.

The introduction of garbage collection has a non-trivial impact on our compiler passes. We introduce a new compiler pass named `expose_allocation` that elaborates the code for allocating tuples. We also make significant changes to `select_instructions`, `build_interference`, `allocate_registers`, and `prelude_and_conclusion` and make minor changes in several more passes.

The following program will serve as our running example. It creates two tuples, one nested inside the other. Both tuples have length one. The program accesses the element in the inner tuple.

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

6.3 Shrink

Recall that the `shrink` pass translates the primitives operators into a smaller set of primitives. This pass comes after type checking and the type checker adds a `HasType` AST node around each `vector` AST node, so you'll need to add a case for `HasType` to the `shrink` pass.

6.4 Expose Allocation

The pass `expose_allocation` lowers tuple creation into a conditional call to the collector followed by allocating the appropriate amount of memory and initializing it. We choose to place the `expose_allocation` pass before `remove_complex_operands` because it generates code that contains complex operands.

The output of `expose_allocation` is a language $\mathcal{L}_{\text{Alloc}}$ replaces tuple creation with new lower-level forms that we use in the translation of tuple creation.

$exp ::= \dots \mid (\text{collect } int) \mid (\text{allocate } int \text{ type}) \mid (\text{global-value } name)$

The `(collect n)` form runs the garbage collector, requesting that it make sure that there are n bytes ready to be allocated. During instruction selection, the `(collect n)` form will become a call to the `collect` function in `runtime.c`. The `(allocate n T)` form obtains memory for n elements (and space at the front for the 64 bit tag), but the elements are not initialized. The T parameter is the type of the tuple: `(Vector $type_1 \dots type_n$)` where $type_i$ is the type of the i th element in the tuple. The `(global-value $name$)` form reads the value of a global variable, such as `free_ptr`.

The following shows the transformation of tuple creation into 1) a sequence of temporary variable bindings for the initializing expressions, 2) a conditional call to `collect`, 3) a call to `allocate`, and 4) the initialization of the tuple. The `len` placeholder refers to the length of the tuple and `bytes` is how many total bytes need to be allocated for the tuple, which is 8 for the tag plus `len` times 8.

```
(has-type (vector  $e_0 \dots e_{n-1}$ ) type)
 $\Rightarrow$ 
(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) ... )))) ...)
```

The sequencing of the initializing expressions e_0, \dots, e_{n-1} prior to the `allocate` is important, as they may trigger garbage collection and we cannot have an allocated but uninitialized tuple on the heap during a collection.

Figure 6.10 shows the output of the `expose_allocation` pass on our running example.

```

(vector-ref
 (vector-ref
  (let ([vecinit6
        (let ([_4 (if (< (+ (global-value free_ptr) 16)
                          (global-value fromspace_end))
                  (void)
                  (collect 16)))]
        (let ([alloc2 (allocate 1 (Vector Integer))])
          (let ([_3 (vector-set! alloc2 0 42)])
            alloc2))))))
 (let ([_8 (if (< (+ (global-value free_ptr) 16)
                  (global-value fromspace_end))
              (void)
              (collect 16)))]
  (let ([alloc5 (allocate 1 (Vector (Vector Integer)))]
    (let ([_7 (vector-set! alloc5 0 vecinit6)])
      alloc5))))
 0)
0)

```

Figure 6.10

Output of the `expose_allocation` pass.

<i>atm</i>	$::=$	$(\text{Int } int) \mid (\text{Var } var)$
<i>exp</i>	$::=$	$atm \mid (\text{Prim 'read } ())$ $\mid (\text{Prim '-(atm)}) \mid (\text{Prim '+ (atm atm)}) \mid (\text{Prim '- (atm atm)})$ $\mid (\text{Let } var \text{ exp exp})$
<i>atm</i>	$::=$	$(\text{Bool } bool)$
<i>exp</i>	$::=$	$(\text{Prim not (atm)}) \mid (\text{Prim cmp (atm atm)}) \mid (\text{If exp exp exp})$
<i>atm</i>	$::=$	(Void)
<i>exp</i>	$::=$	$(\text{GetBang } var) \mid (\text{SetBang } var \text{ exp}) \mid (\text{Begin (exp ...) exp})$ $\mid (\text{WhileLoop exp exp})$
<i>exp</i>	$::=$	$(\text{Collect int}) \mid (\text{Allocate int type}) \mid (\text{GlobalValue var})$
$\mathcal{L}_{\text{Alloc}}^{\text{mon}}$	$::=$	(Program '() exp)

Figure 6.11

$\mathcal{L}_{\text{Alloc}}^{\text{mon}}$ is $\mathcal{L}_{\text{Alloc}}$ in monadic normal form.

6.5 Remove Complex Operands

The forms `collect`, `allocate`, and `global_value` should be treated as complex operands. Figure 6.11 shows the grammar for the output language $\mathcal{L}_{\text{Alloc}}^{\text{mon}}$ of this pass, which is $\mathcal{L}_{\text{Alloc}}$ in monadic normal form.

<i>atm</i>	::=	(Int <i>int</i>) (Var <i>var</i>)
<i>exp</i>	::=	<i>atm</i> (Prim 'read ()) (Prim '- (<i>atm</i>)) (Prim '+ (<i>atm</i> <i>atm</i>)) (Prim '- (<i>atm</i> <i>atm</i>))
<i>stmt</i>	::=	(Assign (Var <i>var</i>) <i>exp</i>)
<i>tail</i>	::=	(Return <i>exp</i>) (Seq <i>stmt</i> <i>tail</i>)
<hr/>		
<i>atm</i>	::=	(Bool <i>bool</i>)
<i>cmp</i>	::=	eq? < <= > >=
<i>exp</i>	::=	(Prim 'not (<i>atm</i>)) (Prim 'cmp (<i>atm</i> <i>atm</i>))
<i>tail</i>	::=	(Goto <i>label</i>) (IfStmt (Prim <i>cmp</i> (<i>atm</i> <i>atm</i>)) (Goto <i>label</i>) (Goto <i>label</i>))
<hr/>		
<i>atm</i>	::=	(Void)
<i>stmt</i>	::=	(Prim 'read ())
<hr/>		
<i>exp</i>	::=	(Allocate <i>int</i> <i>type</i>) (Prim vector-ref (<i>atm</i> (Int <i>int</i>))) (Prim vector-set! (<i>atm</i> (Int <i>int</i>) <i>atm</i>)) (Prim vector-length (<i>atm</i>)) (GlobalValue <i>var</i>)
<i>stmt</i>	::=	(Prim vector-set! (<i>atm</i> (Int <i>int</i>) <i>atm</i>)) (Collect <i>int</i>)
\mathcal{C}_{Tup}	::=	(CProgram <i>info</i> ((<i>label</i> . <i>tail</i>) ...))

Figure 6.12

The abstract syntax of \mathcal{C}_{Tup} , extending \mathcal{C}_{\odot} (Figure 5.7).

6.6 Explicate Control and the \mathcal{C}_{Tup} language

The output of `explicate_control` is a program in the intermediate language \mathcal{C}_{Tup} , whose abstract syntax is defined in Figure 6.12. The new expressions of \mathcal{C}_{Tup} include `allocate`, `vector-ref`, and `vector-set!`, and `global_value`. \mathcal{C}_{Tup} also includes the new `collect` statement. The `explicate_control` pass can treat these new forms much like the other forms that we've already encountered. The output of the `explicate_control` pass on the running example is shown on the left-side of Figure 6.15 in the next section.

6.7 Select Instructions and the $\text{x86}_{\text{Global}}$ Language

In this pass we generate x86 code for most of the new operations that were needed to compile tuples, including `Allocate`, `Collect`, and accessing tuple elements. We compile `GlobalValue` to `Global` because the later has a different concrete syntax (see Figures 6.13 and 6.14).

The tuple read and write forms translate into `movq` instructions. (The +1 in the offset is to move past the tag at the beginning of the tuple representation.)

```

lhs = (vector-ref tup n);
 $\Rightarrow$ 
movq tup', %r11
movq 8(n+1)(%r11), lhs'

lhs = (vector-set! tup n rhs);
 $\Rightarrow$ 
movq tup', %r11
movq rhs', 8(n+1)(%r11)
movq $0, lhs'

```

The lhs' , tup' , and rhs' are obtained by translating from \mathcal{C}_{Tup} to x86. The move of tup' to register `r11` ensures that offset expression $8(n+1)(\%r11)$ contains a register operand. This requires removing `r11` from consideration by the register allocating.

Why not use `rax` instead of `r11`? Suppose we instead used `rax`. Then the generated code for tuple assignment would be

```

movq tup', %rax
movq rhs', 8(n+1)(%rax)

```

Next, suppose that rhs' ends up as a stack location, so `patch_instructions` would insert a move through `rax` as follows.

```

movq tup', %rax
movq rhs', %rax
movq %rax, 8(n+1)(%rax)

```

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

The `vector-length` operation should be translated into a sequence of instructions that read the tag of the tuple and extract the six bits that represent the tuple length, which are the bits starting at index 1 and going up to and including bit 6. The x86 instructions `andq` (for bitwise-and) and `sarq` (shift right) can be used to accomplish this.

We compile the `allocate` form to operations on the `free_ptr`, as shown below. This approach is called *inline allocation* as it implements allocation without a function call, by simply bumping the allocation pointer. It is much more efficient than calling a function for each allocation. The address in the `free_ptr` is the next free address in the FromSpace, so we copy it into `r11` 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. We then initialize the *tag* and finally copy the address in `r11` to the left-hand-side. Refer to Figure 6.8 to see how the tag is organized. We recommend using the Racket operations `bitwise-ior` and `arithmetic-shift` to compute the tag during compilation. The type annotation in the `allocate` form is used to determine the pointer mask region of the tag. The addressing mode `free_ptr(%rip)` essentially stands for the address of the `free_ptr` global variable but uses a special instruction-pointer relative addressing mode of the x86-64 processor. In particular, the assembler computes the distance

<i>reg</i>	::=	rsp rbp rax rbx rcx rdx rsi rdi r8 r9 r10 r11 r12 r13 r14 r15
<i>arg</i>	::=	<i>\$int</i> <i>%reg</i> <i>int(%reg)</i>
<i>instr</i>	::=	addq <i>arg</i> , <i>arg</i> subq <i>arg</i> , <i>arg</i> negq <i>arg</i> movq <i>arg</i> , <i>arg</i> pushq <i>arg</i> popq <i>arg</i> callq <i>label</i> retq jmp <i>label</i> <i>label</i> : <i>instr</i>
<i>bytereg</i>	::=	ah al bh bl ch cl dh dl
<i>arg</i>	::=	<i>%bytereg</i>
<i>cc</i>	::=	e ne l le g ge
<i>instr</i>	::=	xorq <i>arg</i> , <i>arg</i> cmpq <i>arg</i> , <i>arg</i> setcc <i>arg</i> movzbq <i>arg</i> , <i>arg</i> jcc <i>label</i>
<i>arg</i>	::=	<i>label(%rip)</i>
<i>x86_{Global}</i>	::=	.globl main main: <i>instr</i> *

Figure 6.13

The concrete syntax of *x86_{Global}* (extends *x86_{ff}* of Figure 4.9).

d between the address of *free_ptr* and where the *rip* would be at that moment and then changes the *free_ptr(%rip)* argument to *d(%rip)*, which at runtime will compute the address of *free_ptr*.

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

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

```
(collect bytes)
⇒
movq %r15, %rdi
movq $bytes, %rsi
callq collect
```

The concrete and abstract syntax of the *x86_{Global}* language is defined in Figures 6.13 and 6.14. It differs from *x86_{ff}* just in the addition of global variables. Figure 6.15 shows the output of the *select_instructions* pass on the running example.

<i>reg</i>	<code>::=</code>	<code>rsp rbp rax rbx rcx rdx rsi rdi </code> <code>r8 r9 r10 r11 r12 r13 r14 r15</code>
<i>arg</i>	<code>::=</code>	<code>(Imm int) (Reg reg) (Deref reg int)</code>
<i>instr</i>	<code>::=</code>	<code>(Instr addq (arg arg)) (Instr subq (arg arg)) (Instr negq (arg))</code> <code> (Instr movq (arg arg)) (Instr pushq (arg)) (Instr popq (arg))</code> <code> (Callq label int) (Retq) (Jump label)</code>
<i>block</i>	<code>::=</code>	<code>(Block info (instr ...))</code>
<i>bytereg</i>	<code>::=</code>	<code>ah al bh bl ch cl dh dl</code>
<i>arg</i>	<code>::=</code>	<code>(ByteReg bytereg)</code>
<i>cc</i>	<code>::=</code>	<code>e l le g ge</code>
<i>instr</i>	<code>::=</code>	<code>(Instr xorq (arg arg)) (Instr cmpq (arg arg))</code> <code> (Instr set (cc arg)) (Instr movzbq (arg arg))</code> <code> (JumpIf 'cc' label)</code>
<i>arg</i>	<code>::=</code>	<code>(Global label)</code>
<i>x86_{Global}</i>	<code>::=</code>	<code>(X86Program info ((label . block) ...))</code>

Figure 6.14

The abstract syntax of *x86_{Global}* (extends *x86_{If}* of Figure 4.10).

```

start:
  tmp9 = (global-value free_ptr);
  tmp0 = (+ tmp9 16);
  tmp1 = (global-value fromspace_end);
  if (< tmp0 tmp1)
    goto block0;
  else
    goto block1;
block0:
  _4 = (void);
  goto block9;
block1:
  (collect 16)
  goto block9;
block9:
  alloc2 = (allocate 1 (Vector Integer));
  _3 = (vector-set! alloc2 0 42);
  vecinit6 = alloc2;
  tmp2 = (global-value free_ptr);
  tmp3 = (+ tmp2 16);
  tmp4 = (global-value fromspace_end);
  if (< tmp3 tmp4)
    goto block7;
  else
    goto block8;
block7:
  _8 = (void);
  goto block6;
block8:
  (collect 16)
  goto block6;
block6:
  alloc5 = (allocate 1 (Vector (Vector Integer)));
  _7 = (vector-set! alloc5 0 vecinit6);
  tmp5 = (vector-ref alloc5 0);
  return (vector-ref tmp5 0);

```

⇒

```

start:
  movq free_ptr(%rip), tmp9
  movq tmp9, tmp0
  addq $16, tmp0
  movq fromspace_end(%rip), tmp1
  cmpq tmp1, tmp0
  jl block0
  jmp block1
block0:
  movq $0, _4
  jmp block9
block1:
  movq %r15, %rdi
  movq $16, %rsi
  callq collect
  jmp block9
block9:
  movq free_ptr(%rip), %r11
  addq $16, free_ptr(%rip)
  movq $3, 0(%r11)
  movq %r11, alloc2
  movq alloc2, %r11
  movq $42, 8(%r11)
  movq $0, _3
  movq alloc2, vecinit6
  movq free_ptr(%rip), tmp2
  movq tmp2, tmp3
  addq $16, tmp3
  movq fromspace_end(%rip), tmp4
  cmpq tmp4, tmp3
  jl block7
  jmp block8
block7:
  movq $0, _8
  jmp block6
block8:
  movq %r15, %rdi
  movq $16, %rsi
  callq collect
  jmp block6
block6:
  movq free_ptr(%rip), %r11
  addq $16, free_ptr(%rip)
  movq $131, 0(%r11)
  movq %r11, alloc5
  movq alloc5, %r11
  movq vecinit6, 8(%r11)
  movq $0, _7
  movq alloc5, %r11
  movq 8(%r11), tmp5
  movq tmp5, %r11
  movq 8(%r11), %rax
  jmp conclusion

```

Figure 6.15

Output of the `explicate_control` (left) and `select_instructions` (right) passes on the running example.

6.8 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 tuples. It will be the responsibility of the register allocator to make sure that:

1. the root stack is used for spilling tuple-typed variables, and
2. if a tuple-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 interference graph, by adding interference edges between the call-live tuple-typed variables and all the callee-saved registers. (They already interfere with the caller-saved registers.) The type information for variables is in the **Program** form, so we recommend adding another parameter to the **build_interference** function to communicate this alist.

The spilling of tuple-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.

6.9 Prelude and Conclusion

Figure 6.16 shows the output of the **prelude_and_conclusion** pass on the running example. In the prelude and conclusion of the **main** function, we allocate space on the root stack to make room for the spills of tuple-typed variables. We do so by bumping the root stack pointer (**r15**) taking care 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 subtract 8 bytes from **r15**.

One issue that deserves special care is that there may be a call to **collect** prior to the initializing assignments for all the variables in the root stack. We do not want the garbage collector to accidentally think that some uninitialized variable is a pointer that needs to be followed. Thus, we zero-out all locations on the root stack in the prelude of **main**. In Figure 6.16, the instruction **movq \$0, 0(%r15)** is sufficient to accomplish this task because there is only one spill. In general, we have to clear as many words as there are spills of tuple-typed variables. The garbage collector tests each root to see if it is null prior to dereferencing it.

Figure 6.17 gives an overview of all the passes needed for the compilation of \mathcal{L}_{Tup} .

```

        .globl main
main:
    pushq %rbp
    movq %rsp, %rbp
    subq $0, %rsp
    movq $65536, %rdi
    movq $65536, %rsi
    callq initialize
    movq rootstack_begin(%rip), %r15
    movq $0, 0(%r15)
    addq $8, %r15
    jmp start
conclusion:
    subq $8, %r15
    addq $0, %rsp
    popq %rbp
    retq

```

Figure 6.16

The prelude and conclusion generated by the `prelude_and_conclusion` pass for the running example.

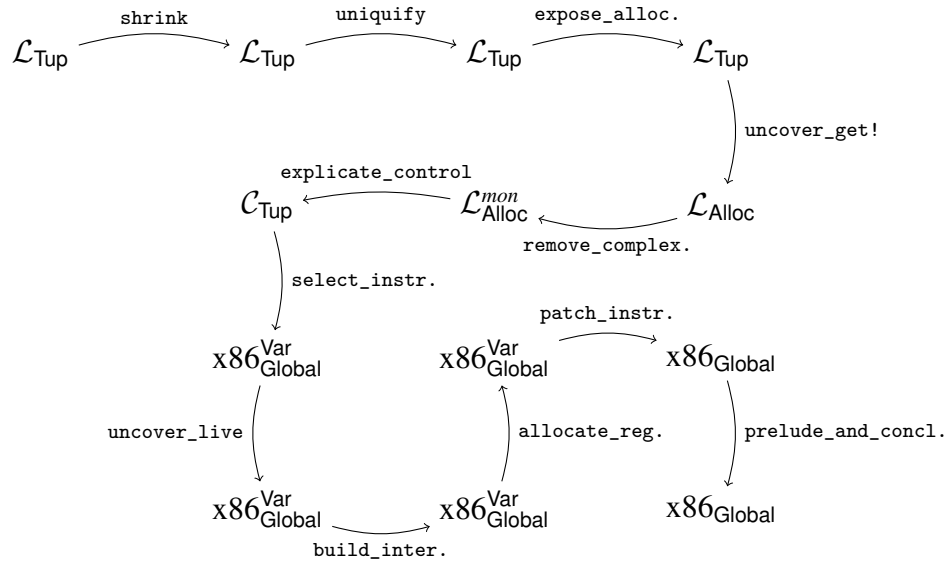
**Figure 6.17**

Diagram of the passes for \mathcal{L}_{Tup} , a language with tuples.

<i>type</i>	::= Integer
<i>exp</i>	::= <i>int</i> (read) (- <i>exp</i>) (+ <i>exp exp</i>) (- <i>exp exp</i>)
<i>exp</i>	::= <i>var</i> (let ([<i>var exp</i>]) <i>exp</i>)
<i>type</i>	::= Boolean
<i>bool</i>	::= #t #f
<i>cmp</i>	::= eq? < <= > >=
<i>exp</i>	::= <i>bool</i> (and <i>exp exp</i>) (or <i>exp exp</i>) (not <i>exp</i>) (<i>cmp exp exp</i>) (if <i>exp exp exp</i>)
<i>type</i>	::= Void
<i>exp</i>	::= (set! <i>var exp</i>) (begin <i>exp* exp</i>) (while <i>exp exp</i>) (void)
<i>type</i>	::= (Vector <i>type*</i>)
<i>exp</i>	::= (vector <i>exp*</i>) (vector-length <i>exp</i>) (vector-ref <i>exp int</i>) (vector-set! <i>exp int exp</i>)
<i>type</i>	::= <i>var</i>
<i>exp</i>	::= (var <i>exp ...</i>)
<i>def</i>	::= (struct <i>var</i> ([<i>var : type</i>] ...) #:mutable)
$\mathcal{L}_{\text{Struct}}$::= <i>def ... exp</i>

Figure 6.18

The concrete syntax of $\mathcal{L}_{\text{Struct}}$, extending $\mathcal{L}_{\text{Tuple}}$ (Figure 6.1).

6.10 Challenge: Simple Structures

The language $\mathcal{L}_{\text{Struct}}$ extends $\mathcal{L}_{\text{Tuple}}$ with support for simple structures. Its concrete syntax is defined in Figure 6.18 and the abstract syntax is in Figure 6.19. Recall that a **struct** in Typed Racket is a user-defined data type that contains named fields and that is heap allocated, similar to a vector. The following is an example of a structure definition, in this case the definition of a **point** type.

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

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

```
(point 7 12)
```

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

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

<i>type</i>	::=	Integer
<i>exp</i>	::=	(Int <i>int</i>) (Prim 'read ()) (Prim '- (<i>exp</i>)) (Prim '+ (<i>exp exp</i>)) (Prim '- (<i>exp exp</i>))
<i>exp</i>	::=	(Var <i>var</i>) (Let <i>var exp exp</i>)
<i>type</i>	::=	Boolean
<i>bool</i>	::=	#t #f
<i>cmp</i>	::=	eq? < <= > >=
<i>op</i>	::=	<i>cmp</i> and or not
<i>exp</i>	::=	(Bool <i>bool</i>) (If <i>exp exp exp</i>)
<i>type</i>	::=	Void
<i>exp</i>	::=	(SetBang <i>var exp</i>) (Begin <i>exp* exp</i>) (WhileLoop <i>exp exp</i>) (Void)
<i>type</i>	::=	(Vector <i>type*</i>)
<i>op</i>	::=	vector vector-length
<i>exp</i>	::=	(Prim vector-ref (<i>exp</i> (Int <i>int</i>))) (Prim vector-set! (<i>exp</i> (Int <i>int</i>) <i>exp</i>)) (HasType <i>exp type</i>)
<i>type</i>	::=	(Var <i>var</i>)
<i>exp</i>	::=	(Apply <i>var exp ...</i>)
<i>def</i>	::=	(StructDef <i>var</i> ([<i>var : type</i>] ...))
$\mathcal{L}_{\text{Struct}}$::=	(ProgramDefsExp '() (def ...) <i>exp</i>)

Figure 6.19

The abstract syntax of $\mathcal{L}_{\text{Struct}}$, extending \mathcal{L}_{Top} (Figure 6.2).

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

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

Exercise 30 Create a type checker for $\mathcal{L}_{\text{Struct}}$ by extending the type checker for \mathcal{L}_{Top} . Extend your compiler with support for simple structures, compiling $\mathcal{L}_{\text{Struct}}$ to x86 assembly code. Create five new test cases that use structures and test your compiler.

<i>type</i>	::=	Integer
<i>exp</i>	::=	<i>int</i> (<i>read</i>) (<i>- exp</i>) (<i>+ exp exp</i>) (<i>- exp exp</i>)
<i>exp</i>	::=	<i>var</i> (<i>let</i> ([<i>var exp</i>]) <i>exp</i>)
<i>type</i>	::=	Boolean
<i>bool</i>	::=	#t #f
<i>cmp</i>	::=	eq? < <= > >=
<i>exp</i>	::=	<i>bool</i> (<i>and exp exp</i>) (<i>or exp exp</i>) (<i>not exp</i>) (<i>cmp exp exp</i>) (<i>if exp exp exp</i>)
<i>type</i>	::=	Void
<i>exp</i>	::=	(<i>set! var exp</i>) (<i>begin exp* exp</i>) (<i>while exp exp</i>) (<i>void</i>)
<i>type</i>	::=	(<i>Vector type*</i>)
<i>exp</i>	::=	(<i>vector exp*</i>) (<i>vector-length exp</i>) (<i>vector-ref exp int</i>) (<i>vector-set! exp int exp</i>)
<i>type</i>	::=	(<i>Vectorof type</i>)
<i>exp</i>	::=	(<i>* exp exp</i>) (<i>make-vector exp exp</i>)
$\mathcal{L}_{\text{While}}^{\text{Array}}$::=	<i>exp</i>

Figure 6.20

The concrete syntax of $\mathcal{L}_{\text{While}}^{\text{Array}}$, extending $\mathcal{L}_{\text{While}}$ (Figure 5.1).

6.11 Challenge: Arrays

In Chapter 6 we studied tuples, that is, a heterogeneous sequences of elements whose length is determined at compile-time. This challenge is also about sequences, but this time the length is determined at run-time and all the elements have the same type (they are homogeneous). We use the term “array” for this later kind of sequence.

The Racket language does not distinguish between tuples and arrays, they are both represented by vectors. However, Typed Racket distinguishes between tuples and arrays: the `Vector` type is for tuples and the `Vectorof` type is for arrays. Figure 6.20 defines the concrete syntax for $\mathcal{L}_{\text{While}}^{\text{Array}}$, extending $\mathcal{L}_{\text{While}}$ with the `Vectorof` type and the `make-vector` primitive operator for creating an array, whose arguments are the length of the array and an initial value for all the elements in the array. The `vector-length`, `vector-ref`, and `vector-ref!` operators that we defined for tuples become overloaded for use with arrays. We also include integer multiplication in $\mathcal{L}_{\text{While}}^{\text{Array}}$, as it is useful in many examples involving arrays such as computing the inner-product of two arrays (Figure 6.21).

The type checker for $\mathcal{L}_{\text{While}}^{\text{Array}}$ is defined in Figure 6.22. The result type of `make-vector` is `(Vectorof T)` where `T` is the type of the initializing expression. The length expression is required to have type `Integer`. The type checking of the operators `vector-length`, `vector-ref`, and `vector-set!` is updated to handle the situation where the vector has type `Vectorof`. In these cases we translate the operators to their `vectorof` form so that later passes can easily distinguish between operations on tuples versus arrays. We override the `operator-types` method to

```

(define (inner-product [A : (Vectorof Integer)] [B : (Vectorof Integer)]
  [n : Integer]) : Integer
  (let ([i 0])
    (let ([prod 0])
      (begin
        (while (< i n)
          (begin
            (set! prod (+ prod (* (vector-ref A i)
                                   (vector-ref B i))))
            (set! i (+ i 1))
          ))
        prod))))

(let ([A (make-vector 2 2)])
  (let ([B (make-vector 2 3)])
    (+ (inner-product A B 2)
      30)))

```

Figure 6.21

Example program that computes the inner-product.

provide the type signature for multiplication: it takes two integers and returns an integer.

The interpreter for $\mathcal{L}_{\text{While}}^{\text{Array}}$ is defined in Figure 6.23. The `make-vector` operator is implemented with Racket's `make-vector` function and multiplication is `fx*`, multiplication for `fixnum` integers.

6.11.1 Data Representation

Just like tuples, we store arrays on the heap which means that the garbage collector will need to inspect arrays. An immediate thought is to use the same representation for arrays that we use for tuples. However, we limit tuples to a length of 50 so that their length and pointer mask can fit into the 64-bit tag at the beginning of each tuple (Section 6.2.3). We intend arrays to allow millions of elements, so we need more bits to store the length. However, because arrays are homogeneous, we only need 1 bit for the pointer mask instead of one bit per array elements. Finally, the garbage collector will need to be able to distinguish between tuples and arrays, so we need to reserve 1 bit for that purpose. So we arrive at the following layout for the 64-bit tag at the beginning of an array:

- The right-most bit is the forwarding bit, just like in a tuple. A 0 indicates it is a forwarding pointer and a 1 indicates it is not.
- The next bit to the left is the pointer mask. A 0 indicates that none of the elements are pointers to the heap and a 1 indicates that all of the elements are pointers.
- The next 61 bits store the length of the array.
- The left-most bit distinguishes between a tuple (0) versus an array (1).


```

(define type-check-Lvecof-class
  (class type-check-Lvec-class
    (super-new)
    (inherit check-type-equal?)

    (define/override (operator-types)
      (append '((* . ((Integer Integer) . Integer)))
        (super operator-types)))

    (define/override (type-check-exp env)
      (lambda (e)
        (define recur (type-check-exp env))
        (match e
          [(Prim 'make-vector (list e1 e2))
           (define-values (e1^ t1) (recur e1))
           (define-values (e2^ elt-type) (recur e2))
           (define vec-type `(Vectorof ,elt-type))
           (values (HasType (Prim 'make-vector (list e1^ e2^)) vec-type)
             vec-type)]
          [(Prim 'vector-ref (list e1 e2))
           (define-values (e1^ t1) (recur e1))
           (define-values (e2^ t2) (recur e2))
           (match* (t1 t2)
             [( `(Vectorof ,elt-type) 'Integer)
              (values (Prim 'vectorof-ref (list e1^ e2^)) elt-type)]
             [(other wise) ((super type-check-exp env) e)]]]
          [(Prim 'vector-set! (list e1 e2 e3) )
           (define-values (e-vec t-vec) (recur e1))
           (define-values (e2^ t2) (recur e2))
           (define-values (e-arg^ t-arg) (recur e3))
           (match t-vec
             [(Vectorof ,elt-type)
              (check-type-equal? elt-type t-arg e)
              (values (Prim 'vectorof-set! (list e-vec e2^ e-arg^)) 'Void)]
             [else ((super type-check-exp env) e)]]]
          [(Prim 'vector-length (list e1))
           (define-values (e1^ t1) (recur e1))
           (match t1
             [(Vectorof ,t)
              (values (Prim 'vectorof-length (list e1^)) 'Integer)]
             [else ((super type-check-exp env) e)]]]
          [else ((super type-check-exp env) e)])))
      ))

(define (type-check-Lvecof p)
  (send (new type-check-Lvecof-class) type-check-program p))

```

Figure 6.22

Type checker for the $\mathcal{L}_{\text{While}}^{\text{Array}}$ language.

In the following subsections we provide hints regarding how to update the passes to handle arrays.

```

(define interp-Lvecof-class
  (class interp-Lvec-class
    (super-new)

    (define/override (interp-op op)
      (match op
        ['make-vector make-vector]
        ['* fx*]
        [else (super interp-op op)]))
  ))

(define (interp-Lvecof p)
  (send (new interp-Lvecof-class) interp-program p))

```

Figure 6.23

Interpreter for $\mathcal{L}_{\text{While}}^{\text{Array}}$.

6.11.2 Bounds Checking

We recommend inserting a new pass named `check_bounds` that inserts code around each the `vector-ref` and `vector-set!` operation to ensure that the index is greater than or equal to zero and less than the `vector-length`.

6.11.3 Expose Allocation

This pass should translate the `make-vector` operator into lower-level operations. In particular, the new AST node (`AllocateArray exp type`) is analogous to the `Allocate` AST node for tuples. It allocates an array of the length specified by the `exp`, but does not initialize the elements of the array. The `type` argument must be `(Vectorof T)` where `T` is the element type for the array. Regarding the initialization of the array, we recommend generated a `while` loop that uses `vector-set!` to put the initializing value into every element of the array.

6.11.4 Remove Complex Operands

Add cases in the `rco_atom` and `rco_exp` for `AllocateArray`. In particular, an `AllocateArray` node is complex and its subexpression must be atomic.

6.11.5 Explicate Control

Add cases for `AllocateArray` to `explicate_tail` and `explicate_assign`.

6.11.6 Select Instructions

Generate instructions for `AllocateArray` similar to those for `Allocate` in Section 6.7 except that the tag at the front of the array should instead use the representation discussed in Section 6.11.1.

Regarding `vectorof-length`, extract the length from the tag according to the representation discussed in Section 6.11.1.

The instructions generated for `vectorof-ref` differ from those for `vector-ref` (Section 6.7) in that the index is not a constant so the offset must be computed at runtime. The same is true for `vectorof-set!`. Also, the `vectorof-set!` may

appear in an assignment and as a stand-alone statement, so make sure to handle both situations in this pass.

Exercise 31 Implement a compiler for the $\mathcal{L}_{\text{While}}^{\text{Array}}$ language by extending your compiler for $\mathcal{L}_{\text{While}}$. Test your compiler on a half dozen new programs, including the one in Figure 6.21 and also a program that multiplies two matrices. Note that although matrices are 2-dimensional arrays, they can be encoded into 1-dimensional arrays by laying out each row in the array, one after the next.

6.12 Challenge: Generational Collection

The copying collector described in Section 6.2 can incur significant runtime overhead because the call to `collect` takes time proportional to all of the live data. One way to reduce this overhead is to reduce how much data is inspected in each call to `collect`. In particular, researchers have observed that recently allocated data is more likely to become garbage than data that has survived one or more previous calls to `collect`. This insight motivated the creation of *generational garbage collectors* that 1) segregates data according to its age into two or more generations, 2) allocates less space for younger generations, so collecting them is faster, and more space for the older generations, and 3) performs collection on the younger generations more frequently than for older generations (Wilson 1992).

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

1. Copy the young generation's FromSpace to its ToSpace then switch the role of the ToSpace and FromSpace
2. If there is enough space for the requested number of bytes in the young FromSpace, then return from `collect`.
3. If there is not enough space in the young FromSpace for the requested bytes, then move the data from the young generation to the old one with the following steps:
 - i. If there is enough room in the old FromSpace, copy the young FromSpace to the old FromSpace and then return.
 - ii. If there is not enough room in the old FromSpace, then collect the old generation by copying the old FromSpace to the old ToSpace and swap the roles of the old FromSpace and ToSpace.
 - iii. If there is enough room now, copy the young FromSpace to the old FromSpace and return. Otherwise, allocate a larger FromSpace and ToSpace for the old generation. Copy the young FromSpace and the old FromSpace into the larger FromSpace for the old generation and then return.

We recommend that you generalize the `cheney` function so that it can be used for all the copies mentioned above: between the young FromSpace and ToSpace, between the old FromSpace and ToSpace, and between the young FromSpace

and old FromSpace. This can be accomplished by adding parameters to `cheney` that replace its use of the global variables `fromspace_begin`, `fromspace_end`, `tospace_begin`, and `tospace_end`.

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

Exercise 32 Adapt the `collect` function in `runtime.c` to implement generational garbage collection, as outlined in this section. Update the code generation for `vector-set!` to implement pointer recording. Make sure that your new compiler and runtime passes your test suite.

6.13 Further Reading

Appel (1990) describes many data representation approaches, including the ones used in the compilation of Standard ML.

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

7 Functions

This chapter studies the compilation of a subset of Typed Racket in which only top-level function definitions are allowed. This kind of function appears in the C programming language and it serves as an important stepping stone to implementing lexically-scoped functions in the form of `lambda` abstractions, which is the topic of Chapter 8.

7.1 The \mathcal{L}_{Fun} Language

The concrete and abstract syntax for function definitions and function application is shown in Figures 7.1 and 7.2, where we define the \mathcal{L}_{Fun} language. Programs in \mathcal{L}_{Fun} begin with zero or more function definitions. The function names from these definitions are in-scope for the entire program, including all of the function definitions (so the ordering of function definitions does not matter). The concrete syntax for function application is `(exp exp ...)` where the first expression must evaluate to a function and the remaining expressions are the arguments. The abstract syntax for function application is `(Apply exp exp*)`.

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

$$(type_1 \cdots type_n \rightarrow type_r)$$

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 \mathcal{L}_{Fun} prevents function definitions from being nested inside each other.

The program in Figure 7.3 is a representative example of defining and using functions in \mathcal{L}_{Fun} . We define a function `map` that applies some other function `f` to both elements of a tuple and returns a new tuple containing the results. We also define a function `inc`. The program applies `map` to `inc` and `(vector 0 41)`. The result is `(vector 1 42)`, from which we return `42`.

The definitional interpreter for \mathcal{L}_{Fun} is in Figure 7.4. The case for the `ProgramDefsExp` AST is responsible for setting up the mutual recursion between

<i>type</i>	::=	Integer
<i>exp</i>	::=	<i>int</i> (<i>read</i>) (<i>- exp</i>) (<i>+ exp exp</i>) (<i>- exp exp</i>)
<i>exp</i>	::=	<i>var</i> (<i>let</i> ([<i>var exp</i>]) <i>exp</i>)
<i>type</i>	::=	Boolean
<i>bool</i>	::=	#t #f
<i>cmp</i>	::=	eq? < <= > >=
<i>exp</i>	::=	<i>bool</i> (<i>and exp exp</i>) (<i>or exp exp</i>) (<i>not exp</i>) (<i>cmp exp exp</i>) (<i>if exp exp exp</i>)
<i>type</i>	::=	Void
<i>exp</i>	::=	(<i>set!</i> <i>var exp</i>) (<i>begin exp* exp</i>) (<i>while exp exp</i>) (<i>void</i>)
<i>type</i>	::=	(<i>Vector type*</i>)
<i>exp</i>	::=	(<i>vector exp*</i>) (<i>vector-length exp</i>) (<i>vector-ref exp int</i>) (<i>vector-set!</i> <i>exp int exp</i>)
<i>type</i>	::=	(<i>type ... -> type</i>)
<i>exp</i>	::=	(<i>exp exp ...</i>)
<i>def</i>	::=	(<i>define</i> (<i>var [var:type]</i> ...) : <i>type exp</i>)
\mathcal{L}_{Fun}	::=	<i>def ... exp</i>

Figure 7.1

The concrete syntax of \mathcal{L}_{Fun} , extending \mathcal{L}_{Typ} (Figure 6.1).

<i>type</i>	::=	Integer
<i>op</i>	::=	read + -
<i>exp</i>	::=	(<i>Int int</i>) (<i>Prim op (exp ...)</i>)
<i>exp</i>	::=	(<i>Var var</i>) (<i>Let var exp exp</i>)
<i>type</i>	::=	Boolean
<i>bool</i>	::=	#t #f
<i>cmp</i>	::=	eq? < <= > >=
<i>op</i>	::=	<i>cmp</i> and or not
<i>exp</i>	::=	(<i>Bool bool</i>) (<i>If exp exp exp</i>)
<i>type</i>	::=	Void
<i>exp</i>	::=	(<i>SetBang var exp</i>) (<i>Begin exp* exp</i>) (<i>WhileLoop exp exp</i>) (<i>Void</i>)
<i>type</i>	::=	(<i>Vector type*</i>)
<i>op</i>	::=	vector vector-length
<i>exp</i>	::=	(<i>Prim vector-ref (exp (Int int))</i>) (<i>Prim vector-set!</i> (<i>exp (Int int) exp</i>)) (<i>HasType exp type</i>)
<i>type</i>	::=	(<i>type ... -> type</i>)
<i>exp</i>	::=	(<i>Apply exp exp ...</i>)
<i>def</i>	::=	(<i>Def var ([var:type] ...) type '() exp</i>)
\mathcal{L}_{Fun}	::=	(<i>ProgramDefsExp '() (def ...) exp</i>)

Figure 7.2

The abstract syntax of \mathcal{L}_{Fun} , extending \mathcal{L}_{Typ} (Figure 6.2).

```

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

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

(vector-ref (map inc (vector 0 41)) 1)

```

Figure 7.3

Example of using functions in \mathcal{L}_{Fun} .

the top-level function definitions. We use the classic back-patching approach that uses mutable variables and makes two passes over the function definitions (Kelsey, Clinger, and (eds.) 1998). In the first pass we set up the top-level environment using a mutable cons cell for each function definition. Note that the `lambda` value for each function is incomplete; it does not yet include the environment. Once the top-level environment is constructed, we then iterate over it and update the `lambda` values to use the top-level environment. To interpret a function application, we match the result of the function expression to obtain a function value. We then extend the function's environment with the mapping of parameters to argument values. Finally, we interpret the body of the function in this extended environment.

The type checker for \mathcal{L}_{Fun} is in Figure 7.5. Similar to the interpreter, the case for the `ProgramDefsExp` AST is responsible for setting up the mutual recursion between the top-level function definitions. We begin by create a mapping `env` from every function name to its type. We then type check the program using this mapping. In the case for function application, we match the type of the function expression to a function type and check that the types of the argument expressions are equal to the function's parameter types. The type of the application as a whole is the return type from the function type.

```

(define interp-Lfun-class
  (class interp-Lvec-class
    (super-new)

    (define/override ((interp-exp env) e)
      (define recur (interp-exp env))
      (match e
        [(Apply fun args)
         (define fun-val (recur fun))
         (define arg-vals (for/list ([e args]) (recur e)))
         (match fun-val
           [(function (,xs ...) ,body ,fun-env)
            (define params-args (for/list ([x xs] [arg arg-vals])
                                     (cons x (box arg))))
            (define new-env (append params-args fun-env))
            ((interp-exp new-env) body)]
           [else (error 'interp-exp "expected function, not ~a" fun-val)])])
        [else ((super interp-exp env) e)]
      ))

    (define/public (interp-def d)
      (match d
        [(Def f (list `[ ,xs : ,ps] ...) rt _ body)
         (cons f (box `(function ,xs ,body ())))])])

    (define/override (interp-program p)
      (match p
        [(ProgramDefsExp info ds body)
         (let ([top-level (for/list ([d ds]) (interp-def d))])
           (for/list ([f (in-dict-values top-level)])
             (set-box! f (match (unbox f)
                               [(function ,xs ,body ())
                                `(function ,xs ,body ,top-level)])))
           ((interp-exp top-level) body))])])
      ))

    (define (interp-Lfun p)
      (send (new interp-Lfun-class) interp-program p))
  )

```

Figure 7.4

Interpreter for the \mathcal{L}_{Fun} language.


```

(define type-check-Lfun-class
  (class type-check-Lvec-class
    (super-new)
    (inherit check-type-equal?)

    (define/public (type-check-apply env e es)
      (define-values (e^ ty) ((type-check-exp env) e))
      (define-values (e* ty*) (for/lists (e* ty*) ([e (in-list es)])
                                           ((type-check-exp env) e)))

      (match ty
        [^(,ty^* ... -> ,rt)
         (for ([arg-ty ty*] [param-ty ty^*])
           (check-type-equal? arg-ty param-ty (Apply e es)))
         (values e^ e* rt))])

    (define/override (type-check-exp env)
      (lambda (e)
        (match e
          [(FunRef f n)
           (values (FunRef f n) (dict-ref env f))]
          [(Apply e es)
           (define-values (e^ es^ rt) (type-check-apply env e es))
           (values (Apply e^ es^) rt)]
          [(Call e es)
           (define-values (e^ es^ rt) (type-check-apply env e es))
           (values (Call e^ es^) rt)]
          [else ((super type-check-exp env) e)])))

    (define/public (type-check-def env)
      (lambda (e)
        (match e
          [(Def f (and p:t* (list ^[,xs : ,ps] ...)) rt info body)
           (define new-env (append (map cons xs ps) env))
           (define-values (body^ ty^) ((type-check-exp new-env) body))
           (check-type-equal? ty^ rt body)
           (Def f p:t* rt info body^))])))

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

    (define/override (type-check-program e)
      (match e
        [(ProgramDefsExp info ds body)
         (define env (for/list ([d ds])
                               (cons (Def-name d) (fun-def-type d))))
         (define ds^ (for/list ([d ds]) ((type-check-def env) d)))
         (define-values (body^ ty) ((type-check-exp env) body))
         (check-type-equal? ty 'Integer body)
         (ProgramDefsExp info ds^ body^)])))

    (define (type-check-Lfun p)
      (send (new type-check-Lfun-class) type-check-program p))

```

Figure 7.5

Type checker for the \mathcal{L}_{Fun} language.

7.2 Functions in x86

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

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

Recall from Section 6.7 that `inc(%rip)` is an example of instruction-pointer relative addressing. It computes the address of `inc`.

In Section 2.2 we used the `callq` instruction to jump to functions whose locations were given by a label, such as `read_int`. To support function calls in this chapter we instead will be jumping to functions whose location are given by an address in a register, that is, we shall use *indirect function calls*. The x86 syntax for this is a `callq` instruction but with an asterisk before the register name.

```
callq *%rbx
```

7.2.1 Calling Conventions

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

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

Regarding (1) parameter passing, recall that the x86-64 calling convention for Unix-based system uses the following six registers to pass arguments to a function, in this order.

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

If there are more than six arguments, then the calling convention mandates to use space on the frame of the caller for the rest of the arguments. However, to ease the implementation of efficient tail calls (Section 7.2.2), we arrange never to need more than six arguments. The return value of the function is stored in register `rax`.

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

Recall that we store variables of tuple type on the root stack. So the prelude of a function needs to move the root stack pointer `r15` up according to the number

Caller View	Callee View	Contents	Frame
8(%rbp)		return address	Caller
0(%rbp)		old <code>rbp</code>	
-8(%rbp)		callee-saved 1	
...		...	
-8j(%rbp)		callee-saved <i>j</i>	
-8(j+1)(%rbp)		local variable 1	
...		...	
-8(j+k)(%rbp)		local variable <i>k</i>	
	8(%rbp)	return address	Callee
	0(%rbp)	old <code>rbp</code>	
	-8(%rbp)	callee-saved 1	
	
	-8n(%rbp)	callee-saved <i>n</i>	
	-8(n+1)(%rbp)	local variable 1	
	
	-8(n+m)(%rbp)	local variable <i>m</i>	

Figure 7.6

Memory layout of caller and callee frames.

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

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

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

Recall that the base pointer, register `rbp`, is used as a point-of-reference within a frame, so that each local variable can be accessed at a fixed offset from the base pointer (Section 2.2). Figure 7.6 shows the general layout of the caller and callee frames.

7.2.2 Efficient Tail Calls

In general, the amount of stack space used by a program is determined by the longest chain of nested function calls. That is, if function f_1 calls f_2 , f_2 calls f_3 , and so on to f_n , then the amount of stack space is linear in n . The depth n can grow quite large if functions are recursive. However, in some cases we can arrange to use only a constant amount of space for a long chain of nested function calls.

A *tail call* is a function call that happens as the last action in a function body. For example, in the following program, the recursive call to `tail_sum` is a tail call.

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

(+ (tail_sum 3 0) 36)
```

At a tail call, the frame of the caller is no longer needed, so we can pop the caller's frame before making the tail call. With this approach, a recursive function that only makes tail calls ends up using a constant amount of stack space. Functional languages like Racket rely heavily on recursive functions, so the definition of Racket *requires* that all tail calls be optimized in this way.

Some care is needed with regards to argument passing in tail calls. As mentioned above, for arguments beyond the sixth, the convention is to use space in the caller's frame for passing arguments. But for a tail call we pop the caller's frame and can no longer use it. An alternative is to use space in the callee's frame for passing arguments. However, this option is also problematic because the caller and callee's frames overlap in memory. As we begin to copy the arguments from their sources in the caller's frame, the target locations in the callee's frame might collide with the sources for later arguments! We solve this problem by using the heap instead of the stack for passing more than six arguments (Section 7.5).

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

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

```
jmp *%rax
```

7.3 Shrink \mathcal{L}_{Fun}

The **shrink** pass performs a minor modification to ease the later passes. This pass introduces an explicit **main** function that gobbles up all the top-level statements of the module. It also changes the top **ProgramDefsExp** form to **ProgramDefs**.

$$\begin{aligned} & (\text{ProgramDefsExp } \text{info } (\text{def } \dots) \text{exp}) \\ \Rightarrow & (\text{ProgramDefs } \text{info } (\text{def } \dots \text{mainDef})) \end{aligned}$$

where *mainDef* is

$$(\text{Def 'main' } () \text{ 'Integer' } () \text{exp'})$$

7.4 Reveal Functions and the $\mathcal{L}_{\text{FunRef}}$ language

The syntax of \mathcal{L}_{Fun} is inconvenient for purposes of compilation in that it conflates the use of function names and local variables. This is a problem because we need to compile the use of a function name differently than the use of a local variable. In particular, we use **leaq** to convert the function name (a label in x86) to an address in a register. Thus, we create a new pass that changes function references from $(\text{Var } f)$ to $(\text{FunRef } f \ n)$ where n is the arity of the function. This pass is named **reveal_functions** and the output language is $\mathcal{L}_{\text{FunRef}}$.

Placing this pass after **uniquify** will make sure that there are no local variables and functions that share the same name. The **reveal_functions** pass should come before the **remove_complex_operands** pass because function references should be categorized as complex expressions.

7.5 Limit Functions

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

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

Each function definition with seven or more parameters is transformed as follows.

$$\begin{aligned} & (\text{Def } f \ ([x_1:T_1] \ \dots \ [x_n:T_n]) \ T_r \ \text{info } \text{body}) \\ \Rightarrow & (\text{Def } f \ ([x_1:T_1] \ \dots \ [x_5:T_5] \ [\text{tup} : (\text{Vector } T_6 \dots T_n)]) \ T_r \ \text{info } \text{body}') \end{aligned}$$

where the *body* is transformed into *body'* by replacing the occurrences of each parameter x_i where $i > 5$ with the k th element of the tuple, where $k = i - 6$.

$$(\text{Var } x_i) \Rightarrow (\text{Prim 'vector-ref' (list tup (Int } k)))$$

For function calls with too many arguments, the **limit_functions** pass transforms them in the following way.

$$(e_0 \ e_1 \ \dots \ e_n) \quad \Rightarrow \quad (e_0 \ e_1 \ \dots \ e_5 \ (\mathbf{vector} \ e_6 \ \dots \ e_n))$$

7.6 Remove Complex Operands

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

The output of this pass, $\mathcal{L}_{\text{FunRef}}^{\text{mon}}$, extends $\mathcal{L}_{\text{Alloc}}^{\text{mon}}$ (Figure 6.11) with **FunRef** and **Apply** in the grammar for expressions.

7.7 Explicate Control and the \mathcal{C}_{Fun} language

Figure 7.7 defines the abstract syntax for \mathcal{C}_{Fun} , the output of **explicate_control**. The auxiliary functions for assignment and tail contexts should be updated with cases for **Apply** and **FunRef** and the function for predicate context should be updated for **Apply** but not **FunRef**. (A **FunRef** cannot be a Boolean.) In assignment and predicate contexts, **Apply** becomes **Call**, whereas in tail position **Apply** becomes **TailCall**. We recommend defining a new auxiliary function for processing function definitions. This code is similar to the case for **Program** in \mathcal{L}_{Top} . The top-level **explicate_control** function that handles the **ProgramDefs** form of \mathcal{L}_{Fun} can then apply this new function to all the function definitions.

<i>atm</i>	$::=$	$(\text{Int } int) \mid (\text{Var } var)$
<i>exp</i>	$::=$	$atm \mid (\text{Prim 'read } ()) \mid (\text{Prim '- } (atm))$ $\mid (\text{Prim '+ } (atm \ atm)) \mid (\text{Prim '- } (atm \ atm))$
<i>stmt</i>	$::=$	$(\text{Assign } (\text{Var } var) \ exp)$
<i>tail</i>	$::=$	$(\text{Return } exp) \mid (\text{Seq } stmt \ tail)$
<hr/>		
<i>atm</i>	$::=$	$(\text{Bool } bool)$
<i>cmp</i>	$::=$	$eq? \mid < \mid \leq \mid > \mid \geq$
<i>exp</i>	$::=$	$(\text{Prim 'not } (atm)) \mid (\text{Prim 'cmp } (atm \ atm))$
<i>tail</i>	$::=$	$(\text{Goto } label)$ $\mid (\text{IfStmt } (\text{Prim } cmp \ (atm \ atm)) \ (\text{Goto } label) \ (\text{Goto } label))$
<hr/>		
<i>atm</i>	$::=$	(Void)
<i>stmt</i>	$::=$	$(\text{Prim 'read } ())$
<hr/>		
<i>exp</i>	$::=$	$(\text{Allocate } int \ type)$ $\mid (\text{Prim vector-ref } (atm \ (\text{Int } int)))$ $\mid (\text{Prim vector-set! } (atm \ (\text{Int } int) \ atm))$ $\mid (\text{Prim vector-length } (atm))$ $\mid (\text{GlobalValue } var)$
<i>stmt</i>	$::=$	$(\text{Prim vector-set! } (atm \ (\text{Int } int) \ atm))$ $\mid (\text{Collect } int)$
<hr/>		
<i>exp</i>	$::=$	$(\text{FunRef } label \ int) \mid (\text{Call } atm \ (atm \ \dots))$
<i>tail</i>	$::=$	$(\text{TailCall } atm \ atm \ \dots)$
<i>def</i>	$::=$	$(\text{Def } label \ ([var: type] \ \dots) \ type \ info \ ((label \ . \ tail) \ \dots))$
\mathcal{C}_{Fun}	$::=$	$(\text{ProgramDefs } info \ (def \ \dots))$

Figure 7.7

The abstract syntax of \mathcal{C}_{Fun} , extending \mathcal{C}_{Top} (Figure 6.12).

```

    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 |
              pushq arg | popq arg | callq label | retq | jmp label |
              label: instr
    -----
    bytereg ::= ah | al | bh | bl | ch | cl | dh | dl
    arg      ::= %bytereg
    cc       ::= e | ne | l | le | g | ge
    instr    ::= xorq arg, arg | cmpq arg, arg | setcc arg | movzbq arg, arg
              | jcc label
    -----
    arg      ::= label(%rip)
    instr    ::= callq *arg | tailjmp arg | leaq arg, %reg
    block    ::= instr+
    def      ::= .globl .align 8 label (label: block)*
    x86callq* ::= def*

```

Figure 7.8

The concrete syntax of $x86_{\text{callq}*}$ (extends $x86_{\text{Global}}$ of Figure 6.13).

7.8 Select Instructions and the $x86_{\text{callq}*}$ Language

The output of select instructions is a program in the $x86_{\text{callq}*}$ language, whose concrete syntax is defined in Figure 7.8 and abstract syntax is defined in Figure 7.9. We use the **align** directive on the labels of function definitions to make sure the bottom three bits are zero, which we make use of in Chapter 9. We discuss the new instructions as needed in this section.

An assignment of a function reference to a variable becomes a load-effective-address instruction as follows, where lhs' is the translation of lhs from atm in \mathcal{C}_{Fun} to arg in $x86_{\text{callq}*}^{\text{Var}}$. The **FunRef** becomes a **Global** AST node, whose concrete syntax is instruction-pointer relative addressing.

$$lhs = (\text{fun-ref } f \ n); \quad \Rightarrow \quad \text{leaq } f(\%rip), lhs'$$

Regarding function definitions, we need to remove the parameters and instead perform parameter passing using the conventions discussed in Section 7.2. That is, the arguments are passed in registers. We recommend turning the parameters into local variables and generating instructions at the beginning of the function to move from the argument passing registers (Section 7.2.1) to these local variables.

```

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

The basic blocks B' are the same as B except that the **start** block is modified to add the instructions for moving from the argument registers to the parameter

<i>reg</i>	::=	rsp rbp rax rbx rcx rdx rsi rdi r8 r9 r10 r11 r12 r13 r14 r15
<i>arg</i>	::=	(Imm <i>int</i>) (Reg <i>reg</i>) (Deref <i>reg int</i>)
<i>instr</i>	::=	(Instr addq (<i>arg arg</i>)) (Instr subq (<i>arg arg</i>)) (Instr negq (<i>arg</i>)) (Instr movq (<i>arg arg</i>)) (Instr pushq (<i>arg</i>)) (Instr popq (<i>arg</i>)) (Callq <i>label int</i>) (Retq) (Jump <i>label</i>)
<i>block</i>	::=	(Block info (<i>instr ...</i>))
<i>bytereg</i>	::=	ah al bh bl ch cl dh dl
<i>arg</i>	::=	(ByteReg <i>bytereg</i>)
<i>cc</i>	::=	e l le g ge
<i>instr</i>	::=	(Instr xorq (<i>arg arg</i>)) (Instr cmpq (<i>arg arg</i>)) (Instr set (<i>cc arg</i>)) (Instr movzbq (<i>arg arg</i>)) (JumpIf ' <i>cc</i> ' <i>label</i>)
<i>arg</i>	::=	(Global <i>label</i>)
<i>instr</i>	::=	(IndirectCallq <i>arg int</i>) (TailJump <i>arg int</i>) (Instr 'leaq (<i>arg</i> (Reg <i>reg</i>)))
<i>block</i>	::=	(Block info (<i>instr ...</i>))
<i>def</i>	::=	(Def <i>label</i> '() type info ((<i>label . block</i>) ...))
<i>x86_{callq}*</i>	::=	(ProgramDefs info (<i>def ...</i>))

Figure 7.9

The abstract syntax of *x86_{callq}** (extends *x86_{Global}* of Figure 6.14).

variables. So the **start** block of *B* shown on the left is changed to the code on the right.

<pre> start: <i>instr</i>₁ ... <i>instr</i>_{<i>n</i>} </pre>	⇒	<pre> start: movq %rdi, <i>x</i>₁ movq %rsi, <i>x</i>₂ ... <i>instr</i>₁ ... <i>instr</i>_{<i>n</i>} </pre>
---	---	---

The interpreter for *x86_{callq}** needs to know how many parameters the function expects, but the parameters are no longer in the syntax of function definitions. Instead, add an entry to *info* that maps **num-params** to the number of parameters to construct *info'*.

By changing the parameters to local variables, we are giving the register allocator control over which registers or stack locations to use for them. If you implemented the move-biasing challenge (Section 3.7), the register allocator will try to assign the parameter variables to the corresponding argument register, in which case the **patch_instructions** pass will remove the **movq** instruction. This happens in the example translation in Figure 7.11 of Section 7.12, in the **add** function. Also, note that the register allocator will perform liveness analysis on this sequence of move instructions and build the interference graph. So, for example, *x*₁ will be marked

as interfering with `rsi` and that will prevent the assignment of x_1 to `rsi`, which is good, because that would overwrite the argument that needs to move into x_2 .

Next, consider the compilation of function calls. In the mirror image of the handling of parameters in function definitions, the arguments are moved to the argument passing registers. Note that the function is not given as a label, but its address is produced by the argument `arg0`. So we translate the call into an indirect function call. The return value from the function is stored in `rax`, so it needs to be moved into the `lhs`.

```
lhs = (Call arg0 arg1 arg2 ...)
⇒
movq arg1, %rdi
movq arg2, %rsi
:
callq *arg0
movq %rax, lhs
```

The `IndirectCallq` AST node includes an integer for the arity of the function, i.e., the number of parameters. That information is useful in the `uncover_live` pass for determining which argument-passing registers are potentially read during the call.

For tail calls, the parameter passing is the same as non-tail calls: generate instructions to move the arguments into the argument passing registers. After that we need to pop the frame from the procedure call stack. However, we do not yet know how big the frame is; that gets determined during register allocation. So instead of generating those instructions here, we invent a new instruction that means “pop the frame and then do an indirect jump”, which we name `TailJump`. The abstract syntax for this instruction includes an argument that specifies where to jump and an integer that represents the arity of the function being called.

Recall that we use the label `start` for the initial block of a program, and in Section 2.7 we recommend labeling the conclusion of the program with `conclusion`, so that `(Return Arg)` can be compiled to an assignment to `rax` followed by a jump to `conclusion`. With the addition of function definitions, there is a start block and conclusion for each function, but their labels need to be unique. We recommend prepending the function’s name to `start` and `conclusion`, respectively, to obtain unique labels.

7.9 Register Allocation

The addition of functions requires some changes to all three aspects of register allocation, which we discuss in the following subsections.

7.9.1 Liveness Analysis

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

Regarding the set of read locations R , the arity field of `TailJump` and `IndirectCallq` determines how many of the argument-passing registers should be considered as read by those instructions. Also, the target field of `TailJump` and `IndirectCallq` should be included in the set of read locations R .

7.9.2 Build Interference Graph

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

Recall that in Section 6.8 we discussed the need to spill tuple-typed variables that are live during a call to `collect`, the garbage collector. With the addition of functions to our language, we need to revisit this issue. Functions that perform allocation contain calls to the collector. Thus, we should not only spill a tuple-typed variable when it is live during a call to `collect`, but we should spill the variable if it is live during call to any user-defined function. Thus, in the `build_interference` pass, we recommend adding interference edges between call-live tuple-typed variables and the callee-saved registers (in addition to the usual addition of edges between call-live variables and the caller-saved registers).

7.9.3 Allocate Registers

The primary change to the `allocate_registers` pass is adding an auxiliary function for handling definitions (the *def* non-terminal in Figure 7.9) with one case for function definitions. The logic is the same as described in Chapter 3, except now register allocation is performed many times, once for each function definition, instead of just once for the whole program.

7.10 Patch Instructions

In `patch_instructions`, you should deal with the x86 idiosyncrasy that the destination argument of `leaq` must be a register. Additionally, you should ensure that the argument of `TailJump` is `rax`, our reserved register—because we trample many other registers before the tail call (as explained in the next section).

7.11 Prelude and Conclusion

Now that register allocation is complete, we can translate the `TailJump` into a sequence of instructions. A naive translation of `TailJump` would simply be `jmp *arg`. However, before the jump we need to pop the current frame to achieve efficient tail calls. This sequence of instructions is the same as the code for the conclusion of a function, except the `retq` is replaced with `jmp *arg`.

Regarding function definitions, we generate a prelude and conclusion for each one. This code is similar to the prelude and conclusion generated for the `main` function in Chapter 6. To review, the prelude of every function should carry out the following steps.

1. Push `rbp` to the stack and set `rbp` to current stack pointer.

2. Push to the stack all of the callee-saved registers that were used for register allocation.
3. Move the stack pointer `rsp` down to make room for the regular spills. (Aligned to 16 bytes.)
4. Move the root stack pointer `r15` up by the size of the root-stack frame for this function, which depends on the number of spilled tuple-typed variables.
5. Initialize to zero all new entries in the root-stack frame.
6. Jump to the start block.

The prelude of the `main` function has an additional task: call the `initialize` function to set up the garbage collector and then move the value of the global `rootstack_begin` in `r15`. This initialization should happen before step 4 above, which depends on `r15`.

The conclusion of every function should do the following.

1. Move the stack pointer back up past the regular spills.
2. Restore the callee-saved registers by popping them from the stack.
3. Move the root stack pointer back down by the size of the root-stack frame for this function.
4. Restore `rbp` by popping it from the stack.
5. Return to the caller with the `retq` instruction.

Exercise 33 Expand your compiler to handle \mathcal{L}_{Fun} as outlined in this chapter. Create 8 new programs that use functions, including examples that pass functions and return functions from other functions, recursive functions, functions that create vectors, and functions that make tail calls. Test your compiler on these new programs and all of your previously created test programs.

Figure 7.10 gives an overview of the passes for compiling \mathcal{L}_{Fun} to x86.

7.12 An Example Translation

Figure 7.11 shows an example translation of a simple function in \mathcal{L}_{Fun} to x86. The figure also includes the results of the `explicate_control` and `select_instructions` passes.

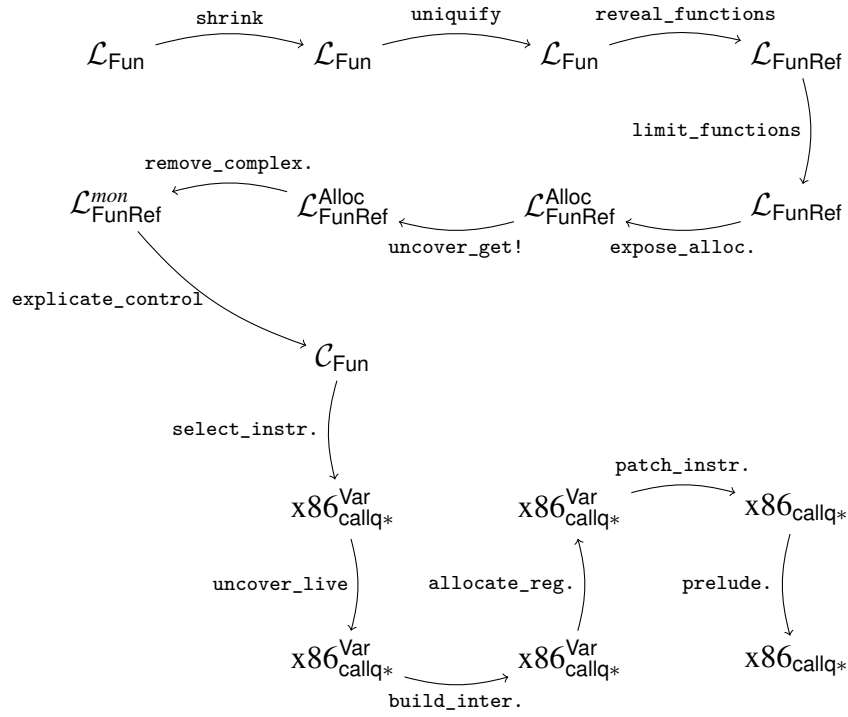
**Figure 7.10**

Diagram of the passes for \mathcal{L}_{Fun} , a language with functions.

```

(define (add [x : Integer]
            [y : Integer])
  : Integer
  (+ x y))

(add 40 2)
↓
(define (add86 [x87 : Integer]
              [y88 : Integer])
  : Integer
  add86start:
  return (+ x87 y88);
)

(define (main) : Integer ()
  mainstart:
  tmp89 = (fun-ref add86 2);
  (tail-call tmp89 40 2)
)

⇒

(define (add86) : Integer
  add86start:
  movq %rdi, x87
  movq %rsi, y88
  movq x87, %rax
  addq y88, %rax
  jmp incl389conclusion
)

(define (main) : Integer
  mainstart:
  leaq (fun-ref add86 2), tmp89
  movq $40, %rdi
  movq $2, %rsi
  tail-jmp tmp89
)
↓

.globl main
.align 8
main:
pushq %rbp
movq %rsp, %rbp
movq $16384, %rdi
movq $16384, %rsi
callq initialize
movq rootstack_begin(%rip), %r15
jmp mainstart

add86:
pushq %rbp
movq %rsp, %rbp
jmp add86start

add86start:
movq %rdi, %rax
addq %rsi, %rax
jmp add86conclusion

add86conclusion:
popq %rbp
retq

mainconclusion:
popq %rbp
retq

```

Figure 7.11

Example compilation of a simple function to x86.

8 Lexically Scoped Functions

This chapter studies lexically scoped functions. Lexical scoping means 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 8.1 written in \mathcal{L}_λ , which extends \mathcal{L}_{Fun} with the `lambda` form for creating lexically scoped functions. The body of the `lambda` refers to three variables: `x`, `y`, and `z`. The binding sites for `x` and `y` are outside of the `lambda`. Variable `y` is bound by the enclosing `let` and `x` is a parameter of function `f`. Note that function `f` returns the `lambda` as its result value. The main expression of the program includes two calls to `f` with different arguments for `x`, first 5 then 3. The functions returned from `f` are bound to variables `g` and `h`. Even though these two functions were created by the same `lambda`, they are really different functions because they use different values for `x`. Applying `g` to 11 produces 20 whereas applying `h` to 15 produces 22, so the result of the program is 42.

The approach that we take for implementing lexically scoped functions is to compile them into top-level function definitions, translating from \mathcal{L}_λ into \mathcal{L}_{Fun} . However, the compiler must give special treatment to variable occurrences such as `x` and `y` in the body of the `lambda` of Figure 8.1. After all, an \mathcal{L}_{Fun} function may not refer to variables defined outside of it. To identify such variable occurrences, we review the standard notion of free variable.

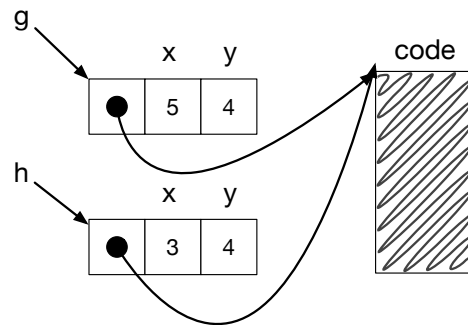
Definition 2 *A variable is **free in expression** e if the variable occurs inside e but does not have an enclosing definition that is also in e .*

```
(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 8.1

Example of a lexically scoped function.

**Figure 8.2**

Flat closure representations for the two functions produced by the `lambda` in Figure 8.1.

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

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

So the free variables of a `lambda` are the ones that need special treatment. We need to transport, at runtime, the values of those variables from the point where the `lambda` was created to the point where the `lambda` is applied. An efficient solution to the problem, due to Cardelli (1983), is to bundle the values of the free variables together with a function pointer into a tuple, an arrangement called a *flat closure* (which we shorten to just “closure”). By design, we have all the ingredients to make closures: Chapter 6 gave us tuples and Chapter 7 gave us function pointers. The function pointer resides at index 0 and the values for the free variables fill in the rest of the tuple.

Let us revisit the example in Figure 8.1 to see how closures work. It is a three-step dance. The program calls function `f`, which creates a closure for the `lambda`. The closure is a tuple whose first element is a pointer to the top-level function that we will generate for the `lambda`, the second element is the value of `x`, which is 5, and the third element is 4, the value of `y`. The closure does not contain an element for `z` because `z` is not a free variable of the `lambda`. Creating the closure is step 1 of the dance. The closure is returned from `f` and bound to `g`, as shown in Figure 8.2. The second call to `f` creates another closure, this time with 3 in the second slot (for `x`). This closure is also returned from `f` but bound to `h`, which is also shown in Figure 8.2.

Continuing with the example, consider the application of `g` to 11 in Figure 8.1. To apply a closure, we obtain the function pointer from the first element of the closure and call it, passing in the closure itself and then the regular arguments, in this case 11. This technique for applying a closure is step 2 of the dance. But doesn't this `lambda` only take 1 argument, for parameter `z`? The third and final step of the dance is generating a top-level function for a `lambda`. We add an additional parameter for the closure and we insert an initialization at the beginning of the function for each

<i>type</i>	::=	Integer
<i>exp</i>	::=	<i>int</i> (<i>read</i>) (<i>- exp</i>) (<i>+ exp exp</i>) (<i>- exp exp</i>)
<i>exp</i>	::=	<i>var</i> (<i>let</i> ([<i>var exp</i>]) <i>exp</i>)
<i>type</i>	::=	Boolean
<i>bool</i>	::=	#t #f
<i>cmp</i>	::=	eq? < <= > >=
<i>exp</i>	::=	<i>bool</i> (<i>and exp exp</i>) (<i>or exp exp</i>) (<i>not exp</i>) (<i>cmp exp exp</i>) (<i>if exp exp exp</i>)
<i>type</i>	::=	Void
<i>exp</i>	::=	(<i>set!</i> <i>var exp</i>) (<i>begin exp* exp</i>) (<i>while exp exp</i>) (<i>void</i>)
<i>type</i>	::=	(<i>Vector type*</i>)
<i>exp</i>	::=	(<i>vector exp*</i>) (<i>vector-length exp</i>) (<i>vector-ref exp int</i>) (<i>vector-set!</i> <i>exp int exp</i>)
<i>type</i>	::=	(<i>type ... -> type</i>)
<i>exp</i>	::=	(<i>exp exp ...</i>)
<i>def</i>	::=	(<i>define</i> (<i>var [var:type] ...</i>) : <i>type exp</i>)
<i>exp</i>	::=	(<i>lambda:</i> ([<i>var:type</i>] ...) : <i>type exp</i>) (<i>procedure-arity exp</i>)
\mathcal{L}_λ	::=	<i>def ... exp</i>

Figure 8.3

The concrete syntax of \mathcal{L}_λ , extending \mathcal{L}_{Fun} (Figure 7.1) with **lambda**.

free variable, to bind those variables to the appropriate elements from the closure parameter. This three-step dance is known as *closure conversion*. We discuss the details of closure conversion in Section 8.4 and show the code generated from the example in Section 8.5. But first we define the syntax and semantics of \mathcal{L}_λ in Section 8.1.

8.1 The \mathcal{L}_λ Language

The concrete and abstract syntax for \mathcal{L}_λ , a language with anonymous functions and lexical scoping, is defined in Figures 8.3 and 8.4. It adds the **lambda** form to the grammar for \mathcal{L}_{Fun} , which already has syntax for function application. The **procedure-arity** operation returns the number of parameters of a given function, an operation that we need for the translation of dynamic typing in Chapter 9.

Figure 8.5 shows the definitional interpreter for \mathcal{L}_λ . The case for **Lambda** saves the current environment inside the returned function value. Recall that during function application, the environment stored in the function value, extended with the mapping of parameters to argument values, is used to interpret the body of the function.

Figure 8.6 shows how to type check the new **lambda** form. The body of the **lambda** is checked in an environment that includes the current environment (because it is lexically scoped) and also includes the **lambda**'s parameters. We require the body's type to match the declared return type.

<i>type</i>	::=	Integer
<i>op</i>	::=	read + -
<i>exp</i>	::=	(Int <i>int</i>) (Prim <i>op</i> (<i>exp</i> ...))
<i>exp</i>	::=	(Var <i>var</i>) (Let <i>var exp exp</i>)
<i>type</i>	::=	Boolean
<i>bool</i>	::=	#t #f
<i>cmp</i>	::=	eq? < <= > >=
<i>op</i>	::=	<i>cmp</i> and or not
<i>exp</i>	::=	(Bool <i>bool</i>) (If <i>exp exp exp</i>)
<i>type</i>	::=	Void
<i>exp</i>	::=	(SetBang <i>var exp</i>) (Begin <i>exp* exp</i>) (WhileLoop <i>exp exp</i>) (Void)
<i>type</i>	::=	(Vector <i>type*</i>)
<i>op</i>	::=	vector vector-length
<i>exp</i>	::=	(Prim vector-ref (<i>exp</i> (Int <i>int</i>)))
		(Prim vector-set! (<i>exp</i> (Int <i>int</i>) <i>exp</i>))
		(HasType <i>exp type</i>)
<i>type</i>	::=	(<i>type</i> ... -> <i>type</i>)
<i>exp</i>	::=	(Apply <i>exp exp</i> ...)
<i>def</i>	::=	(Def <i>var</i> ([<i>var:type</i>] ...) <i>type</i> '() <i>exp</i>)
<i>exp</i>	::=	(Lambda ([<i>var:type</i>] ...) <i>type exp</i>)
<i>op</i>	::=	procedure-arity
\mathcal{L}_λ	::=	(ProgramDefsExp '() (<i>def</i> ...) <i>exp</i>)

Figure 8.4

The abstract syntax of \mathcal{L}_λ , extending \mathcal{L}_{Fun} (Figure 7.2).

```

(define interp-Lambda-class
  (class interp-Lfun-class
    (super-new)

    (define/override (interp-op op)
      (match op
        ['procedure-arity
         (lambda (v)
           (match v
            [(function (,xs ...) ,body ,lam-env) (length xs)]
            [else (error 'interp-op "expected a function, not ~a" v)]))]
        [else (super interp-op op)]))

    (define/override ((interp-exp env) e)
      (define recur (interp-exp env))
      (match e
        [(Lambda (list `[,xs : ,Ts] ...) rT body)
         `(function ,xs ,body ,env)]
        [else ((super interp-exp env) e)]))
      ))

(define (interp-Lambda p)
  (send (new interp-Lambda-class) interp-program p))

```

Figure 8.5Interpreter for \mathcal{L}_λ .

```

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

```

Figure 8.6Type checking \mathcal{L}_λ .

8.2 Assignment and Lexically Scoped Functions

The combination of lexically-scoped functions and assignment to variables raises a challenge with the flat-closure approach to implementing lexically-scoped functions. Consider the following example in which function `f` has a free variable `x` that is changed after `f` is created but before the call to `f`.

```
(let ([x 0])
  (let ([y 0])
    (let ([z 20])
      (let ([f (lambda: ([a : Integer]) : Integer (+ a (+ x z)))]
        (begin
          (set! x 10)
          (set! y 12)
          (f y)))))))
```

The correct output for this example is 42 because the call to `f` is required to use the current value of `x` (which is 10). Unfortunately, the closure conversion pass (Section 8.4) generates code for the `lambda` that copies the old value of `x` into a closure. Thus, if we naively apply closure conversion, the output of this program would be 32.

A first attempt at solving this problem would be to save a pointer to `x` in the closure and change the occurrences of `x` inside the `lambda` to dereference the pointer. Of course, this would require assigning `x` to the stack and not to a register. However, the problem goes a bit deeper. Consider the following example that returns a function that refers to a local variable of the enclosing function.

```
(define (f []) : Integer
  (let ([x 0])
    (let ([g (lambda: () : Integer x)])
      (begin
        (set! x 42)
        g))))
((f))
```

In this example, the lifetime of `x` extends beyond the lifetime of the call to `f`. Thus, if we were to store `x` on the stack frame for the call to `f`, it would be gone by the time we call `g`, leaving us with dangling pointers for `x`. This example demonstrates that when a variable occurs free inside a function, its lifetime becomes indefinite. Thus, the value of the variable needs to live on the heap. The verb *box* is often used for allocating a single value on the heap, producing a pointer, and *unbox* for dereferencing the pointer. We introduce a new pass named `convert_assignments` to address this challenge.

8.3 Assignment Conversion

The purpose of the `convert_assignments` pass is to address the challenge regarding the interaction between variable assignments and closure conversion. First we

identify which variables need to be boxed, then we transform the program to box those variables. In general, boxing introduces runtime overhead that we would like to avoid, so we should box as few variables as possible. We recommend boxing the variables in the intersection of the following two sets of variables:

1. The variables that are free in a `lambda`.
2. The variables that appear on the left-hand side of an assignment.

The first condition is a must but the second condition is conservative. It is possible to develop a more liberal condition using static program analysis.

Consider again the first example from Section 8.2:

```
(let ([x 0])
  (let ([y 0])
    (let ([z 20])
      (let ([f (lambda: ([a : Integer]) : Integer (+ a (+ x z)))]
        (begin
          (set! x 10)
          (set! y 12)
          (f y)))))))
```

The variables `x` and `y` are assigned-to. The variables `x` and `z` occur free inside the `lambda`. Thus, variable `x` needs to be boxed but not `y` or `z`. The boxing of `x` consists of three transformations: initialize `x` with a tuple whose elements are uninitialized, replace reads from `x` with tuple reads, and replace each assignment to `x` with a tuple write. The output of `convert_assignments` for this example is as follows.

```
(define (main) : Integer
  (let ([x0 (vector 0)])
    (let ([y1 0])
      (let ([z2 20])
        (let ([f4 (lambda: ([a3 : Integer]) : Integer
          (+ a3 (+ (vector-ref x0 0) z2)))]
          (begin
            (vector-set! x0 0 10)
            (set! y1 12)
            (f4 y1)))))))
```

To compute the free variables of all the `lambda` expressions, we recommend defining two auxiliary functions:

1. `free_variables` computes the free variables of an expression, and
2. `free_in_lambda` collects all of the variables that are free in any of the `lambda` expressions, using `free_variables` in the case for each `lambda`.

To compute the variables that are assigned-to, we recommend using the `collect-set!` function that we introduced in Section 5.4, but updated to include the new AST forms such as `Lambda`.

Let AF be the intersection of the set of variables that are free in a `lambda` and that are assigned-to in the enclosing function definition.

Next we discuss the `convert_assignments` pass. In the case for `(Var x)`, if x is in AF , then unbox it by translating `(Var x)` to a tuple read.

```
(Var x)
⇒
(Prim 'vector-ref (list (Var x) (Int 0)))
```

In the case for assignment, recursively process the right-hand side rhs to obtain rhs' . If the left-hand side x is in AF , translate the assignment into a tuple-write as follows.

```
(SetBang x rhs)
⇒
(Prim 'vector-set! (list (Var x) (Int 0) rhs'))
```

The case for `Lambda` is non-trivial, but it is similar to the case for function definitions, which we discuss next. To translate a function definition, we first compute AF , the intersection of the variables that are free in a `lambda` and that are assigned-to. We then apply assignment conversion to the body of the function definition. Finally, we box the parameters of this function definition that are in AF . For example, the parameter x of the following function `g` needs to be boxed.

```
(define (g [x : Integer]) : Integer
  (let ([f (lambda: ([a : Integer]) : Integer (+ a x))])
    (begin
      (set! x 10)
      (f 32))))
```

We box parameter x by creating a local variable named `x` that is initialized to a tuple whose contents is the value of the parameter, which has been renamed to `x_0`.

```
(define (g [x_0 : Integer]) : Integer
  (let ([x (vector x_0)])
    (let ([f (lambda: ([a : Integer]) : Integer
      (+ a (vector-ref x 0)))]
      (begin
        (vector-set! x 0 10)
        (f 32)))))
```

8.4 Closure Conversion

The compiling of lexically-scoped functions into top-level function definitions and flat closures is accomplished in the pass `convert_to_closures` that comes after `reveal_functions` and before `limit_functions`.

As usual, we implement the pass as a recursive function over the AST. The interesting cases are the ones for `lambda` and function application. We transform a `lambda` expression into an expression that creates a closure, that is, a tuple whose first element is a function pointer and the rest of the elements are the values of the free variables of the `lambda`. However, we use the `Closure` AST node instead of using a tuple so that we can record the arity. In the generated code below, fvs is

the free variables of the lambda and *name* is a unique symbol generated to identify the lambda. The *arity* is the number of parameters (the length of *ps*).

```
(Lambda ps rt body)
⇒
(Closure arity (cons (FunRef name arity) fvs))
```

In addition to transforming each **Lambda** AST node into a tuple, we create a top-level function definition for each **Lambda**, as shown below.

```
(Def name ([clos : (Vector _ fvs ...)] ps' ...) rt'
  (Let fvs1 (Prim 'vector-ref (list (Var clos) (Int 1)))
    ...
    (Let fvsn (Prim 'vector-ref (list (Var clos) (Int n)))
      body')...))
```

The **clos** parameter refers to the closure. Translate the type annotations in *ps* and the return type *rt*, as discussed in the next paragraph, to obtain *ps'* and *rt'*. The type *closTy* is a tuple type whose first element type is **_** (the dummy type) and the rest of the element types are the types of the free variables in the lambda. We use **_** because it is non-trivial to give a type to the function in the closure's type.¹⁴ The free variables become local variables that are initialized with their values in the closure.

Closure conversion turns every function into a tuple, so the type annotations in the program must also be translated. We recommend defining an auxiliary recursive function for this purpose. Function types should be translated as follows.

```
(T1, ..., Tn -> Tr)
⇒
(Vector ((Vector) T'1, ..., T'n -> T'r))
```

The above type says that the first thing in the tuple is a function. The first parameter of the function is a tuple (a closure) and the rest of the parameters are the ones from the original function, with types *T'₁*, ..., *T'_n*. The type for the closure omits the types of the free variables because 1) those types are not available in this context and 2) we do not need them in the code that is generated for function application. So this type only describes the first component of the closure tuple. At runtime the tuple may have more components, but we ignore them at this point.

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

```
(Apply e es)
⇒
(Let tmp e'
  (Apply (Prim 'vector-ref (list (Var tmp) (Int 0))) (cons (Var tmp) es'))))
```

14. To give an accurate type to a closure, we would need to add existential types to the type checker (Minamide, Morrisett, and Harper 1996).

```

(define (f6 [x7 : Integer]) : (Integer -> Integer)
  (let ([y8 4])
    (lambda: ([z9 : Integer]) : Integer
      (+ x7 (+ y8 z9)))))

(define (main) : Integer
  (let ([g0 ((fun-ref f6 1) 5)])
    (let ([h1 ((fun-ref f6 1) 3)])
      (+ (g0 11) (h1 15)))))

⇒

(define (f6 [fvs4 : _] [x7 : Integer]) : (Vector ((Vector _) Integer -> Integer))
  (let ([y8 4])
    (closure 1 (list (fun-ref lambda2 1) x7 y8))))

(define (lambda2 [fvs3 : (Vector _ Integer Integer)] [z9 : Integer]) : Integer
  (let ([x7 (vector-ref fvs3 1)])
    (let ([y8 (vector-ref fvs3 2)])
      (+ x7 (+ y8 z9)))))

(define (main) : Integer
  (let ([g0 (let ([clos5 (closure 1 (list (fun-ref f6 1)))]
                ((vector-ref clos5 0) clos5 5)))]
    (let ([h1 (let ([clos6 (closure 1 (list (fun-ref f6 1)))]
                  ((vector-ref clos6 0) clos6 3)))]
      (+ ((vector-ref g0 0) g0 11) ((vector-ref h1 0) h1 15)))))

```

Figure 8.7

Example of closure conversion.

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

$$(\text{FunRef } f \ n) \quad \Rightarrow \quad (\text{Closure } n \ (\text{FunRef } f \ n) \ '())$$

We no longer need the annotated assignment statement **AnnAssign** to support the type checking of **lambda** expressions, so we translate it to a regular **Assign** statement.

The top-level function definitions need to be updated to take an extra closure parameter but that parameter is ignored in the body of those functions.

8.5 An Example Translation

Figure 8.7 shows the result of **reveal_functions** and **convert_to_closures** for the example program demonstrating lexical scoping that we discussed at the beginning of this chapter.

Exercise 34 Expand your compiler to handle \mathcal{L}_λ as outlined in this chapter. Create 5 new programs that use **lambda** functions and make use of lexical scoping. Test your compiler on these new programs and all of your previously created test programs.

<i>atm</i>	::=	(Int <i>int</i>) (Var <i>var</i>)
<i>exp</i>	::=	<i>atm</i> (Prim 'read ()) (Prim '- (<i>atm</i>))
		(Prim '+ (<i>atm atm</i>)) (Prim '- (<i>atm atm</i>))
<i>stmt</i>	::=	(Assign (Var <i>var</i>) <i>exp</i>)
<i>tail</i>	::=	(Return <i>exp</i>) (Seq <i>stmt tail</i>)
<hr/>		
<i>atm</i>	::=	(Bool <i>bool</i>)
<i>cmp</i>	::=	eq? < <= > >=
<i>exp</i>	::=	(Prim 'not (<i>atm</i>)) (Prim 'cmp (<i>atm atm</i>))
<i>tail</i>	::=	(Goto <i>label</i>)
		(IfStmt (Prim <i>cmp</i> (<i>atm atm</i>)) (Goto <i>label</i>) (Goto <i>label</i>))
<hr/>		
<i>atm</i>	::=	(Void)
<i>stmt</i>	::=	(Prim 'read ())
<hr/>		
<i>exp</i>	::=	(Allocate <i>int type</i>)
		(Prim vector-ref (<i>atm</i> (Int <i>int</i>)))
		(Prim vector-set! (<i>atm</i> (Int <i>int</i>) <i>atm</i>))
		(Prim vector-length (<i>atm</i>))
		(GlobalValue <i>var</i>)
<i>stmt</i>	::=	(Prim vector-set! (<i>atm</i> (Int <i>int</i>) <i>atm</i>))
		(Collect <i>int</i>)
<hr/>		
<i>exp</i>	::=	(FunRef <i>label int</i>) (Call <i>atm</i> (<i>atm</i> ...))
<i>tail</i>	::=	(TailCall <i>atm atm</i> ...)
<i>def</i>	::=	(Def <i>label</i> ([<i>var:type</i>] ...) <i>type info</i> ((<i>label . tail</i>) ...))
<hr/>		
<i>exp</i>	::=	(AllocateClosure <i>int type int</i>)
<i>op</i>	::=	procedure-arity
<i>C_{Clos}</i>	::=	(ProgramDefs <i>info def</i> *)

Figure 8.8

The abstract syntax of $\mathcal{C}_{\text{Clos}}$, extending \mathcal{C}_{Fun} (Figure 7.7).

8.6 Expose Allocation

Compile the (Closure *arity exp**) form into code that allocates and initializes a tuple, similar to the translation of the tuple creation in Section 6.4. The only difference is replacing the use of (Allocate *len type*) with (AllocateClosure *len type arity*).

8.7 Explicate Control and $\mathcal{C}_{\text{Clos}}$

The output language of `explicate_control` is $\mathcal{C}_{\text{Clos}}$ whose abstract syntax is defined in Figure 8.8. The only differences with respect to \mathcal{C}_{Fun} is the addition of the `AllocateClosure` form to the grammar for *exp* and the `procedure-arity` operator. The handling of `AllocateClosure` in the `explicate_control` pass is similar to the handling of other expressions such as primitive operators.

8.8 Select Instructions

Compile `(AllocateClosure len type arity)` in almost the same way as the `(Allocate len type)` form (Section 6.7). The only difference is that you should place the *arity* in the tag that is stored at position 0 of the vector. Recall that in Section 6.7 a portion of the 64-bit tag was not used. We store the arity in the 5 bits starting at position 58.

Compile the **procedure-arity** operator into a sequence of instructions that access the tag from position 0 of the vector and extract the 5-bits starting at position 58 from the tag.

Figure 8.9 provides an overview of the passes needed for the compilation of \mathcal{L}_λ .

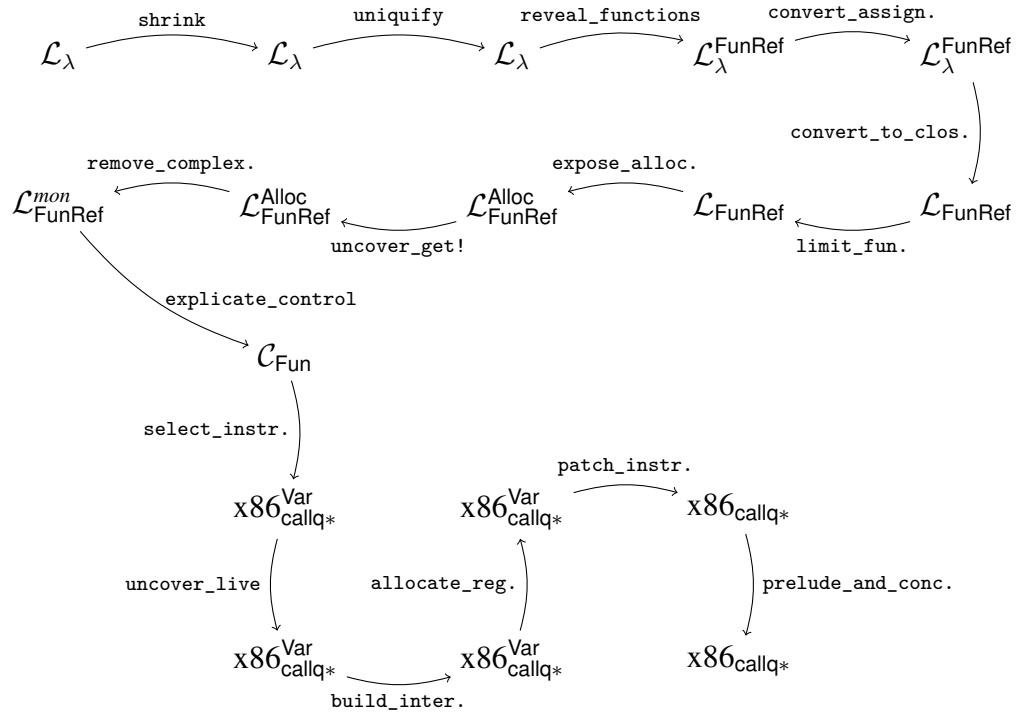
**Figure 8.9**

Diagram of the passes for \mathcal{L}_λ , a language with lexically-scoped functions.

8.9 Challenge: Optimize Closures

In this chapter we compiled lexically-scoped functions into a relatively efficient representation: flat closures. However, even this representation comes with some overhead. For example, consider the following program with a function `tail_sum` that does not have any free variables and where all the uses of `tail_sum` are in applications where we know that only `tail_sum` is being applied (and not any other functions).

```
(define (tail_sum [n : Integer] [s : Integer]) : Integer
  (if (eq? n 0)
      s
      (tail_sum (- n 1) (+ n s))))

(+ (tail_sum 3 0) 36)
```

As described in this chapter, we uniformly apply closure conversion to all functions, obtaining the following output for this program.

```
(define (tail_sum1 [fvs5 : _] [n2 : Integer] [s3 : Integer]) : Integer
  (if (eq? n2 0)
      s3
      (let ([clos4 (closure (list (fun-ref tail_sum1 2)))]
            [(vector-ref clos4 0) clos4 (+ n2 -1) (+ n2 s3)])))

(define (main) : Integer
  (+ (let ([clos6 (closure (list (fun-ref tail_sum1 2)))]
        [(vector-ref clos6 0) clos6 3 0]) 27))
```

In the previous chapter, there would be no allocation in the program and the calls to `tail_sum` would be direct calls. In contrast, the above program allocates memory for each closure and the calls to `tail_sum` are indirect. These two differences incur considerable overhead in a program such as this one, where the allocations and indirect calls occur inside a tight loop.

One might think that this problem is trivial to solve: can't we just recognize calls of the form `(Apply (FunRef f n) args)` and compile them to direct calls instead of treating it like a call to a closure? We would also drop the new `fvs` parameter of `tail_sum`. However, this problem is not so trivial because a global function may “escape” and become involved in applications that also involve closures. Consider the following example in which the application `(f 41)` needs to be compiled into a closure application, because the `lambda` may flow into `f`, but the `inc` function might also flow into `f`.

```

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

(let ([y (read)])
  (let ([f (if (eq? (read) 0)
               inc
               (lambda: ([x : Integer]) : Integer (- x y)))]))
    (f 41)))

```

If a global function name is used in any way other than as the operator in a direct call, then we say that the function *escapes*. If a global function does not escape, then we do not need to perform closure conversion on the function.

Exercise 35 Implement an auxiliary function for detecting which global functions escape. Using that function, implement an improved version of closure conversion that does not apply closure conversion to global functions that do not escape but instead compiles them as regular functions. Create several new test cases that check whether you properly detect whether global functions escape or not.

So far we have reduced the overhead of calling global functions, but it would also be nice to reduce the overhead of calling a `lambda` when we can determine at compile time which `lambda` will be called. We refer to such calls as *known calls*. Consider the following example in which a `lambda` is bound to `f` and then applied.

```

(let ([y (read)])
  (let ([f (lambda: ([x : Integer]) : Integer
                (+ x y))])
    (f 21)))

```

Closure conversion compiles the application `(f 21)` into an indirect call:

```

(define (lambda5 [fvs6 : (Vector _ Integer)] [x3 : Integer]) : Integer
  (let ([y2 (vector-ref fvs6 1)])
    (+ x3 y2)))

(define (main) : Integer
  (let ([y2 (read)])
    (let ([f4 (Closure 1 (list (fun-ref lambda5 1) y2))])
      ((vector-ref f4 0) f4 21))))

```

but we can instead compile the application `(f 21)` into a direct call:

```

(define (main) : Integer
  (let ([y2 (read)])
    (let ([f4 (Closure 1 (list (fun-ref lambda5 1) y2))])
      ((fun-ref lambda5 1) f4 21))))

```

The problem of determining which `lambda` will be called from a particular application is quite challenging in general and the topic of considerable research (Shivers 1988; Gilray et al. 2016). For the following exercise we recommend that you compile an application to a direct call when the operator is a variable and the variable

is `let`-bound to a closure. This can be accomplished by maintaining an environment mapping variables to function names. Extend the environment whenever you encounter a closure on the right-hand side of a `let`, mapping the variable to the name of the global function for the closure. This pass should come after closure conversion.

Exercise 36 Implement a compiler pass, named `optimize_known_calls`, that compiles known calls into direct calls. Verify that your compiler is successful in this regard on several example programs.

These exercises only scratches the surface of optimizing of closures. A good next step for the interested reader is to look at the work of Keep, Hearn, and Dybvig (2012).

8.10 Further Reading

The notion of lexically scoped functions predates modern computers by about a decade. They were invented by Church (1932), who proposed the lambda calculus as a foundation for logic. Anonymous functions were included in the LISP (McCarthy 1960) programming language but were initially dynamically scoped. The Scheme dialect of LISP adopted lexical scoping and Steele (1978) demonstrated how to efficiently compile Scheme programs. However, environments were represented as linked lists, so variable look-up was linear in the size of the environment. Appel (1991) gives a detailed description of several closure representations. In this chapter we represent environments using flat closures, which were invented by Cardelli (1983, 1984) for the purposes of compiling the ML language (Gordon et al. 1978; Milner, Tofte, and Harper 1990). With flat closures, variable look-up is constant time but the time to create a closure is proportional to the number of its free variables. Flat closures were reinvented by Dybvig (1987b) in his Ph.D. thesis and used in Chez Scheme version 1 (Dybvig 2006).

9 Dynamic Typing

In this chapter we learn how to compile \mathcal{L}_{Dyn} , a dynamically typed language that is a subset of Racket. The focus on dynamic typing is in contrast to the previous chapters, which have studied the compilation of statically typed languages. In dynamically typed languages such as \mathcal{L}_{Dyn} , a particular expression may produce a value of a different type each time it is executed. Consider the following example with a conditional `if` 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*, a word composed of the Greek roots “poly”, meaning “many”, and “morph”, meaning “form”. There are several kinds of polymorphism in programming languages, such as subtype polymorphism and parametric polymorphism (Cardelli and Wegner 1985). The kind of polymorphism we study in this chapter does not have a special name but it is the kind that arises in dynamically typed languages.

Another characteristic of dynamically typed languages is that primitive operations, such as `not`, are often defined to operate on many different types of values. In fact, in Racket, the `not` operator produces a result for any kind of value: given `#f` it returns `#t` and given anything else it returns `#f`.

Furthermore, even when primitive operations restrict their inputs to values of a certain type, this restriction is enforced at runtime instead of during compilation. For example, the tuple read operation (`vector-ref #t 0`) results in a run-time error because the first argument must be a tuple, not a Boolean.

9.1 The \mathcal{L}_{Dyn} Language

The concrete and abstract syntax of \mathcal{L}_{Dyn} is defined in Figures 9.1 and 9.2. There is no type checker for \mathcal{L}_{Dyn} because it only checks types at runtime.

The definitional interpreter for \mathcal{L}_{Dyn} is presented in Figure 9.3 and its auxiliary functions are defined in Figure 9.4. Consider the match case for `(Int n)`. Instead of simply returning the integer `n` (as in the interpreter for \mathcal{L}_{Var} in Figure 2.5), the interpreter for \mathcal{L}_{Dyn} creates a *tagged value* that combines an underlying value with a tag that identifies what kind of value it is. We define the following struct to represent tagged values.

<i>type</i>	::=	Integer
<i>exp</i>	::=	<i>int</i> (read) (- <i>exp</i>) (+ <i>exp exp</i>) (- <i>exp exp</i>)
<i>exp</i>	::=	<i>var</i> (let ([<i>var exp</i>]) <i>exp</i>)
<i>type</i>	::=	Boolean
<i>bool</i>	::=	#t #f
<i>cmp</i>	::=	eq? < <= > >=
<i>exp</i>	::=	<i>bool</i> (and <i>exp exp</i>) (or <i>exp exp</i>) (not <i>exp</i>) (<i>cmp exp exp</i>) (if <i>exp exp exp</i>)
<i>type</i>	::=	Void
<i>exp</i>	::=	(set! <i>var exp</i>) (begin <i>exp* exp</i>) (while <i>exp exp</i>) (void)
<i>type</i>	::=	(Vector <i>type*</i>)
<i>exp</i>	::=	(vector <i>exp*</i>) (vector-length <i>exp</i>) (vector-ref <i>exp int</i>) (vector-set! <i>exp int exp</i>)
<i>exp</i>	::=	(<i>exp exp ...</i>) (lambda (<i>var ...</i>) <i>exp</i>) (boolean? <i>exp</i>) (integer? <i>exp</i>) (vector? <i>exp</i>) (procedure? <i>exp</i>) (void? <i>exp</i>)
<i>def</i>	::=	(define (<i>var var ...</i>) <i>exp</i>)
\mathcal{L}_{Dyn}	::=	<i>def ... exp</i>

Figure 9.1

Syntax of \mathcal{L}_{Dyn} , an untyped language (a subset of Racket).

<i>type</i>	::=	Integer
<i>exp</i>	::=	(Int <i>int</i>) (Prim 'read ()) (Prim '- (<i>exp</i>)) (Prim '+ (<i>exp exp</i>)) (Prim '- (<i>exp exp</i>))
<i>exp</i>	::=	(Var <i>var</i>) (Let <i>var exp exp</i>)
<i>type</i>	::=	Boolean
<i>bool</i>	::=	#t #f
<i>cmp</i>	::=	eq? < <= > >=
<i>op</i>	::=	<i>cmp</i> and or not
<i>exp</i>	::=	(Bool <i>bool</i>) (If <i>exp exp exp</i>)
<i>type</i>	::=	Void
<i>exp</i>	::=	(SetBang <i>var exp</i>) (Begin <i>exp* exp</i>) (WhileLoop <i>exp exp</i>) (Void)
<i>type</i>	::=	(Vector <i>type*</i>)
<i>op</i>	::=	vector vector-length
<i>exp</i>	::=	(Prim vector-ref (<i>exp</i> (Int <i>int</i>))) (Prim vector-set! (<i>exp</i> (Int <i>int</i>) <i>exp</i>)) (HasType <i>exp type</i>)
<i>exp</i>	::=	(Apply <i>exp exp ...</i>) (Lambda (<i>var ...</i>) 'Any <i>exp</i>)
<i>def</i>	::=	(Def <i>var</i> (<i>var ...</i>) 'Any '() <i>exp</i>)
\mathcal{L}_{Dyn}	::=	(ProgramDefsExp '() (<i>def ...</i>) <i>exp</i>)

Figure 9.2

The abstract syntax of \mathcal{L}_{Dyn} .


```
(struct Tagged (value tag) #:transparent)
```

The tags are `Integer`, `Boolean`, `Void`, `Vector`, and `Procedure`. Tags are closely related to types but don't always capture all the information that a type does. For example, a vector of type `(Vector Any Any)` is tagged with `Vector` and a procedure of type `(Any Any -> Any)` is tagged with `Procedure`.

Next consider the match case for accessing the element of a tuple. The `check-tag` auxiliary function (Figure 9.4) is used to ensure that the first argument is a tuple and the second is an integer. If they are not, a `trapped-error` is raised. Recall from Section 1.5 that when a definition interpreter raises a `trapped-error` error, the compiled code must also signal an error by exiting with return code 255. A `trapped-error` is also raised if the index is not less than the length of the vector.

```

(define ((interp-Ldyn-exp env) ast)
  (define recur (interp-Ldyn-exp env))
  (match ast
    [(Var x) (dict-ref env x)]
    [(Int n) (Tagged n 'Integer)]
    [(Bool b) (Tagged b 'Boolean)]
    [(Lambda xs rt body)
     (Tagged `(function ,xs ,body ,env) 'Procedure)]
    [(Prim 'vector es)
     (Tagged (apply vector (for/list ([e es]) (recur e))) 'Vector)]
    [(Prim 'vector-ref (list e1 e2))
     (define vec (recur e1)) (define i (recur e2))
     (check-tag vec 'Vector ast) (check-tag i 'Integer ast)
     (unless (< (Tagged-value i) (vector-length (Tagged-value vec)))
       (error 'trapped-error "index ~a too big\nin ~v" (Tagged-value i) ast))
     (vector-ref (Tagged-value vec) (Tagged-value i)))]
    [(Prim 'vector-set! (list e1 e2 e3))
     (define vec (recur e1)) (define i (recur e2)) (define arg (recur e3))
     (check-tag vec 'Vector ast) (check-tag i 'Integer ast)
     (unless (< (Tagged-value i) (vector-length (Tagged-value vec)))
       (error 'trapped-error "index ~a too big\nin ~v" (Tagged-value i) ast))
     (vector-set! (Tagged-value vec) (Tagged-value i) arg)
     (Tagged (void) 'Void)]
    [(Let x e body) ((interp-Ldyn-exp (cons (cons x (recur e)) env)) body)]
    [(Prim 'and (list e1 e2)) (recur (If e1 e2 (Bool #f)))]
    [(Prim 'or (list e1 e2))
     (define v1 (recur e1))
     (match (Tagged-value v1) [#f (recur e2)] [else v1])]
    [(Prim 'eq? (list l r)) (Tagged (equal? (recur l) (recur r)) 'Boolean)]
    [(Prim op (list e1))
     #:when (set-member? type-predicates op)
     (tag-value ((interp-op op) (Tagged-value (recur e1)))))]
    [(Prim op es)
     (define args (map recur es))
     (define tags (for/list ([arg args]) (Tagged-tag arg)))
     (unless (for/or ([expected-tags (op-tags op)])
               (equal? expected-tags tags))
       (error 'trapped-error "illegal argument tags ~a\nin ~v" tags ast))
     (tag-value
      (apply (interp-op op) (for/list ([a args]) (Tagged-value a)))))]
    [(If q t f)
     (match (Tagged-value (recur q)) [#f (recur f)] [else (recur t)])]
    [(Apply f es)
     (define new-f (recur f)) (define args (map recur es))
     (check-tag new-f 'Procedure ast) (define f-val (Tagged-value new-f))
     (match f-val
       [(function ,xs ,body ,lam-env)
        (unless (eq? (length xs) (length args))
          (error 'trapped-error "~a != ~a\nin ~v" (length args) (length xs) ast))
        (define new-env (append (map cons xs args) lam-env))
        ((interp-Ldyn-exp new-env) body)]
       [else (error "interp-Ldyn-exp, expected function, not" f-val)])])

```

Figure 9.3

Interpreter for the \mathcal{L}_{Dyn} language.

```

(define (interp-op op)
  (match op
    ['+ fx+]
    ['- fx-]
    ['read read-fixnum]
    ['not (lambda (v) (match v [#t #f] [#f #t])))]
    ['< (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)]))]
    ['boolean? boolean?]
    ['integer? fixnum?]
    ['void? void?]
    ['vector? vector?]
    ['vector-length vector-length]
    ['procedure? (match-lambda
      [^(functions ,xs ,body ,env) #t] [else #f])]
    [else (error 'interp-op "unknown operator" op)]))

(define (op-tags op)
  (match op
    ['+ '((Integer Integer))]
    ['- '((Integer Integer) (Integer))]
    ['read '()]
    ['not '((Boolean))]
    ['< '((Integer Integer))]
    ['<= '((Integer Integer))]
    ['> '((Integer Integer))]
    ['>= '((Integer Integer))]
    ['vector-length '((Vector))])

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

(define (tag-value v)
  (cond [(boolean? v) (Tagged v 'Boolean)]
        [(fixnum? v) (Tagged v 'Integer)]
        [(procedure? v) (Tagged v 'Procedure)]
        [(vector? v) (Tagged v 'Vector)]
        [(void? v) (Tagged v 'Void)]
        [else (error 'tag-value "unidentified value ~a" v)]))

(define (check-tag val expected ast)
  (define tag (Tagged-tag val))
  (unless (eq? tag expected)
    (error 'trapped-error "expected ~a, not ~a\nin ~v" expected tag ast)))

```

Figure 9.4

Auxiliary functions for the \mathcal{L}_{Dyn} interpreter.

9.2 Representation of Tagged Values

The interpreter for \mathcal{L}_{Dyn} introduced a new kind of value, a tagged value. To compile \mathcal{L}_{Dyn} to x86 we must decide how to represent tagged values at the bit level. Because almost every operation in \mathcal{L}_{Dyn} involves manipulating tagged values, the representation must be efficient. Recall that all of our values are 64 bits. We shall steal the 3 right-most bits to encode the tag. We use 001 to identify integers, 100 for Booleans, 010 for tuples, 011 for procedures, and 101 for the void value. We define the following auxiliary function for mapping types to tag codes.

```

tagof(Integer)=001
tagof(Boolean)=100
tagof((Vector ...))=010
tagof((... -> ...))=011
tagof(Void)=101

```

This stealing of 3 bits comes at some price: integers are now restricted to the range from -2^{60} to 2^{60} . The stealing does not adversely affect tuples 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.

To make tagged values into first-class entities, we can give them a type, called **Any**, and define operations such as **Inject** and **Project** for creating and using them, yielding the statically typed \mathcal{L}_{Any} intermediate language. We describe how to compile \mathcal{L}_{Dyn} to \mathcal{L}_{Any} in Section 9.4 but first we describe the \mathcal{L}_{Any} language in greater detail.

9.3 The \mathcal{L}_{Any} Language

The abstract syntax of \mathcal{L}_{Any} is defined in Figure 9.5. 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 halts the program if the type tag does not match T . Note that in both **Inject** and **Project**, the type T is restricted to a flat type *f*type, which simplifies the implementation and corresponds with the needs for compiling \mathcal{L}_{Dyn} .

The **any-vector** operators adapt the tuple operations so that they can be applied to a value of type **Any**. They also generalize the tuple operations in that the index is not restricted to be a literal integer in the grammar but is allowed to be any expression.

The type predicates such as **boolean?** expect their argument to produce a tagged value; they return **#t** if the tag corresponds to the predicate and they return **#f** otherwise.

<i>type</i>	::=	Integer
<i>op</i>	::=	read + -
<i>exp</i>	::=	(Int <i>int</i>) (Prim <i>op</i> (<i>exp</i> ...))
<i>exp</i>	::=	(Var <i>var</i>) (Let <i>var exp exp</i>)
<i>type</i>	::=	Boolean
<i>bool</i>	::=	#t #f
<i>cmp</i>	::=	eq? < <= > >=
<i>op</i>	::=	<i>cmp</i> and or not
<i>exp</i>	::=	(Bool <i>bool</i>) (If <i>exp exp exp</i>)
<i>type</i>	::=	Void
<i>exp</i>	::=	(SetBang <i>var exp</i>) (Begin <i>exp* exp</i>) (WhileLoop <i>exp exp</i>) (Void)
<i>type</i>	::=	(Vector <i>type*</i>)
<i>op</i>	::=	vector vector-length
<i>exp</i>	::=	(Prim vector-ref (<i>exp</i> (Int <i>int</i>)))
		(Prim vector-set! (<i>exp</i> (Int <i>int</i>) <i>exp</i>))
		(HasType <i>exp type</i>)
<i>type</i>	::=	(<i>type</i> ... -> <i>type</i>)
<i>exp</i>	::=	(Apply <i>exp exp</i> ...)
<i>def</i>	::=	(Def <i>var</i> ([<i>var: type</i>] ...) <i>type</i> '() <i>exp</i>)
<i>exp</i>	::=	(Lambda ([<i>var: type</i>] ...) <i>type exp</i>)
<i>op</i>	::=	procedure-arity
<i>type</i>	::=	Any
<i>ftype</i>	::=	Integer Boolean Void (Vector Any ...) (Any ... -> Any)
<i>op</i>	::=	any-vector-length any-vector-ref any-vector-set!
		boolean? integer? vector? procedure? void?
<i>exp</i>	::=	(Inject <i>exp ftype</i>) (Project <i>exp ftype</i>)
\mathcal{L}_{Any}	::=	(ProgramDefsExp '() (def ...) <i>exp</i>)

Figure 9.5

The abstract syntax of \mathcal{L}_{Any} , extending \mathcal{L}_λ (Figure 8.4).

The type checker for \mathcal{L}_{Any} is shown in Figure 9.6 and uses the auxiliary functions in Figure 9.7. The interpreter for \mathcal{L}_{Any} is in Figure 9.8 and its auxiliary functions are in Figure 9.9.

```

(define type-check-Lany-class
  (class type-check-Llambda-class
    (super-new)
    (inherit check-type-equal?)

    (define/override (type-check-exp env)
      (lambda (e)
        (define recur (type-check-exp env))
        (match e
          [(Inject e1 ty)
            (unless (flat-ty? ty)
              (error 'type-check "may only inject from flat type, not ~a" ty))
            (define-values (new-e1 e-ty) (recur e1))
            (check-type-equal? e-ty ty e)
            (values (Inject new-e1 ty) 'Any)]
          [(Project e1 ty)
            (unless (flat-ty? ty)
              (error 'type-check "may only project to flat type, not ~a" ty))
            (define-values (new-e1 e-ty) (recur e1))
            (check-type-equal? e-ty 'Any e)
            (values (Project new-e1 ty) ty)]
          [(Prim 'any-vector-length (list e1))
            (define-values (e1^ t1) (recur e1))
            (check-type-equal? t1 'Any e)
            (values (Prim 'any-vector-length (list e1^)) 'Integer)]
          [(Prim 'any-vector-ref (list e1 e2))
            (define-values (e1^ t1) (recur e1))
            (define-values (e2^ t2) (recur e2))
            (check-type-equal? t1 'Any e)
            (check-type-equal? t2 'Integer e)
            (values (Prim 'any-vector-ref (list e1^ e2^)) 'Any)]
          [(Prim 'any-vector-set! (list e1 e2 e3))
            (define-values (e1^ t1) (recur e1))
            (define-values (e2^ t2) (recur e2))
            (define-values (e3^ t3) (recur e3))
            (check-type-equal? t1 'Any e)
            (check-type-equal? t2 'Integer e)
            (check-type-equal? t3 'Any e)
            (values (Prim 'any-vector-set! (list e1^ e2^ e3^)) 'Void)]
          [(Prim pred (list e1))
            #:when (set-member? (type-predicates) pred)
            (define-values (new-e1 e-ty) (recur e1))
            (check-type-equal? e-ty 'Any e)
            (values (Prim pred (list new-e1)) 'Boolean)]
          [(Prim 'eq? (list arg1 arg2))
            (define-values (e1 t1) (recur arg1))
            (define-values (e2 t2) (recur arg2))
            (match* (t1 t2)
              [(Vector ,ts1 ...) (Vector ,ts2 ...)] (void)]
              [(other wise) (check-type-equal? t1 t2 e)]
            (values (Prim 'eq? (list e1 e2)) 'Boolean)]
          [else ((super type-check-exp env) e)])))
  ))

```

Figure 9.6

Type checker for the \mathcal{L}_{Any} language.

```

(define/override (operator-types)
  (append
    '((integer? . ((Any) . Boolean))
      (vector? . ((Any) . Boolean))
      (procedure? . ((Any) . Boolean))
      (void? . ((Any) . Boolean)))
    (super operator-types)))

(define/public (type-predicates)
  (set 'boolean? 'integer? 'vector? 'procedure? 'void?))

(define/public (flat-ty? ty)
  (match ty
    [(or `Integer `Boolean `Void) #t]
    [`(Vector ,ts ...) (for/and ([t ts]) (eq? t 'Any))]
    [`(Vectorof Any) #t]
    [`(,ts ... -> ,rt)
     (and (eq? rt 'Any) (for/and ([t ts]) (eq? t 'Any)))]
    [else #f]))

```

Figure 9.7

Auxiliary methods for type checking \mathcal{L}_{Any} .

```

(define interp-Lany-class
  (class interp-Llambda-class
    (super-new)

    (define/override (interp-op op)
      (match op
        ['boolean? (match-lambda
          [^(tagged ,v1 ,tg) (equal? tg (any-tag 'Boolean))]
          [else #f])]
        ['integer? (match-lambda
          [^(tagged ,v1 ,tg) (equal? tg (any-tag 'Integer))]
          [else #f])]
        ['vector? (match-lambda
          [^(tagged ,v1 ,tg) (equal? tg (any-tag `(Vector Any)))]
          [else #f])]
        ['procedure? (match-lambda
          [^(tagged ,v1 ,tg) (equal? tg (any-tag `(Any -> Any)))]
          [else #f])]
        ['eq? (match-lambda*
          [^((tagged ,v1^ ,tg1) (tagged ,v2^ ,tg2))
            (and (eq? v1^ v2^) (equal? tg1 tg2))]
          [ls (apply (super interp-op op) ls)]]]
        ['any-vector-ref (lambda (v i)
          (match v [^(tagged ,v^ ,tg) (vector-ref v^ i)]))]
        ['any-vector-set! (lambda (v i a)
          (match v [^(tagged ,v^ ,tg) (vector-set! v^ i a)]))]
        ['any-vector-length (lambda (v)
          (match v [^(tagged ,v^ ,tg) (vector-length v^)]))]
        [else (super interp-op op)]))

    (define/override ((interp-exp env) e)
      (define recur (interp-exp env))
      (match e
        [(Inject e ty) ^ (tagged ,(recur e) ,(any-tag ty))]
        [(Project e ty2) (apply-project (recur e) ty2)]
        [else ((super interp-exp env) e)]))
    ))

(define (interp-Lany p)
  (send (new interp-Lany-class) interp-program p))

```

Figure 9.8Interpreter for \mathcal{L}_{Any} .


```

(define/public (apply-inject v tg) (Tagged v tg))

(define/public (apply-project v ty2)
  (define tag2 (any-tag ty2))
  (match v
    [(Tagged v1 tag1)
     (cond
      [(eq? tag1 tag2)
       (match ty2
         [(Vector ,ts ...)
          (define l1 ((interp-op 'vector-length) v1))
          (cond
           [(eq? l1 (length ts)) v1]
           [else (error 'apply-project "vector length mismatch, ~a != ~a"
                        l1 (length ts))])]
         [_,(ts ... -> ,rt)
          (match v1
            [(function ,xs ,body ,env)
             (cond [(eq? (length xs) (length ts)) v1]
                   [else
                    (error 'apply-project "arity mismatch ~a != ~a"
                          (length xs) (length ts))])]
            [else (error 'apply-project "expected function not ~a" v1)]]]
         [else (error 'apply-project "tag mismatch ~a != ~a" tag1 tag2)]]]
      [else (error 'apply-project "expected tagged value, not ~a" v)])])

```

Figure 9.9

Auxiliary functions for interpreting \mathcal{L}_{Any} .

<code>#t</code>	\Rightarrow	<code>(inject #t Boolean)</code>
<code>(+ e₁ e₂)</code>	\Rightarrow	<code>(inject (+ (project e'₁ Integer) (project e'₂ Integer)) Integer)</code>
<code>(lambda (x₁ ... x_n) e)</code>	\Rightarrow	<code>(inject (lambda: ([x₁:Any] ... [x_n:Any]):Any e') (Any ... Any -> Any))</code>
<code>(e₀ e₁ ... e_n)</code>	\Rightarrow	<code>((project e'₀ (Any ... Any -> Any)) e'₁ ... e'_n)</code>
<code>(vector-ref e₁ e₂)</code>	\Rightarrow	<code>(any-vector-ref e'₁ e'₂)</code>
<code>(if e₁ e₂ e₃)</code>	\Rightarrow	<code>(if (eq? e'₁ (inject #f Boolean)) e'₃ e'₂)</code>
<code>(eq? e₁ e₂)</code>	\Rightarrow	<code>(inject (eq? e'₁ e'₂) Boolean)</code>
<code>(not e₁)</code>	\Rightarrow	<code>(if (eq? e'₁ (inject #f Boolean)) (inject #t Boolean) (inject #f Boolean))</code>

Figure 9.10
Cast Insertion

9.4 Cast Insertion: Compiling \mathcal{L}_{Dyn} to \mathcal{L}_{Any}

The `cast_insert` pass compiles from \mathcal{L}_{Dyn} to \mathcal{L}_{Any} . Figure 9.10 shows the compilation of many of the \mathcal{L}_{Dyn} forms into \mathcal{L}_{Any} . An important invariant of this pass is that given any subexpression e in the \mathcal{L}_{Dyn} program, the pass will produce an expression e' in \mathcal{L}_{Any} that has type `Any`. For example, the first row in Figure 9.10 shows the compilation of the Boolean `#t`, which must be injected to produce an expression of type `Any`. The second row of Figure 9.10, the compilation of addition, is representative of compilation for many primitive 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 9.10) shows what happens when we need to produce type annotations: we simply use `Any`. The compilation of `if` and `eq?` demonstrate how this pass has to account for some differences in behavior between \mathcal{L}_{Dyn} and \mathcal{L}_{Any} . The \mathcal{L}_{Dyn} language is more permissive than \mathcal{L}_{Any} regarding what kind of values can be used in various places. For example, the condition of an `if` does not have to be a Boolean. For `eq?`, the arguments need not be of the same type (in that case the result is `#f`).

9.5 Reveal Casts

In the `reveal_casts` pass we recommend compiling `Project` into a conditional expression that checks whether the value's tag matches the target type; if it does, the value is converted to a value of the target type by removing the tag; if it does not, the program exits. To perform these actions we need a new primitive operation, `tag-of-any`, and two new forms, `ValueOf` and `Exit`. The `tag-of-any` operation retrieves the type tag from a tagged value of type `Any`. The `ValueOf` form retrieves the underlying value from a tagged value. The `ValueOf` form includes the type for the underlying value which is used by the type checker. Finally, the `Exit` form ends the execution of the program.

If the target type of the projection is `Boolean` or `Integer`, then `Project` can be translated as follows.

```
(Project e ftype)
⇒
(Let tmp e'
  (If (Prim 'eq? (list (Prim 'tag-of-any (list (Var tmp)))
                    (Int tagof(ftype))))
    (ValueOf tmp ftype)
    (Exit)))
```

If the target type of the projection is a tuple or function type, then there is a bit more work to do. For tuples, check that the length of the tuple type matches the length of the tuple. For functions, check that the number of parameters in the function type matches the function's arity.

Regarding `Inject`, we recommend compiling it to a slightly lower-level primitive operation named `make-any`. This operation takes a tag instead of a type.

```
(Inject e ftype)
⇒
(Prim 'make-any (list e' (Int tagof(ftype))))
```

The type predicates (`boolean?`, etc.) can be translated into uses of `tag-of-any` and `eq?` in a similar way as in the translation of `Project`.

The `any-vector-ref` and `any-vector-set!` operations combine the projection action with the vector operation. Also, the read and write operations allow arbitrary expressions for the index so the type checker for \mathcal{L}_{Any} (Figure 9.6) cannot guarantee that the index is within bounds. Thus, we insert code to perform bounds checking at runtime. The translation for `any-vector-ref` is as follows and the other two operations are translated in a similar way.

```

(Prim 'any-vector-ref (list e1 e2))
⇒
(Let v e'1
  (Let i e'2
    (If (Prim 'eq? (list (Prim 'tag-of-any (list (Var v))) (Int 2)))
      (If (Prim '< (list (Var i) (Prim 'any-vector-length (list (Var v)))))
        (Prim 'any-vector-ref (list (Var v) (Var i)))
        (Exit))
      (Exit))))

```

9.6 Remove Complex Operands

The `ValueOf` and `Exit` forms are both complex expressions. The subexpression of `ValueOf` must be atomic.

9.7 Explicate Control and \mathcal{C}_{Any}

The output of `explicate_control` is the \mathcal{C}_{Any} language whose syntax is defined in Figure 9.11. The `ValueOf` form that we added to \mathcal{L}_{Any} remains an expression and the `Exit` expression becomes a *tail*. Also, note that the index argument of `vector-ref` and `vector-set!` is an *atm* instead of an integer, as it was in \mathcal{C}_{Tup} (Figure 6.12).

9.8 Select Instructions

In the `select_instructions` pass we translate the primitive operations on the `Any` type to x86 instructions that manipulate the 3 tag bits of the tagged value. In the following descriptions, given an atom e we use a primed variable e' to refer to the result of translating e into an x86 argument.

make-any We recommend compiling the `make-any` operation as follows if the tag is for `Integer` or `Boolean`. The `salq` instruction shifts the destination to the left by the number of bits specified its source argument (in this case 3, the length of the tag) and it preserves the sign of the integer. We use the `orq` instruction to combine the tag and the value to form the tagged value.

```

(Assign lhs (Prim 'make-any (list e (Int tag))))
⇒
movq e', lhs'
salq $3, lhs'
orq $tag, lhs'

```

The instruction selection for tuples and procedures is different because there is no need to shift them to the left. The rightmost 3 bits are already zeros so we simply combine the value and the tag using `orq`.

<i>atm</i>	<code>::= (Int <i>int</i>) (Var <i>var</i>)</code>
<i>exp</i>	<code>::= <i>atm</i> (Prim 'read ()) (Prim '- (<i>atm</i>))</code> <code> (Prim '+ (<i>atm</i> <i>atm</i>)) (Prim '- (<i>atm</i> <i>atm</i>))</code>
<i>stmt</i>	<code>::= (Assign (Var <i>var</i>) <i>exp</i>)</code>
<i>tail</i>	<code>::= (Return <i>exp</i>) (Seq <i>stmt</i> <i>tail</i>)</code>
<hr/>	
<i>atm</i>	<code>::= (Bool <i>bool</i>)</code>
<i>cmp</i>	<code>::= eq? < <= > >=</code>
<i>exp</i>	<code>::= (Prim 'not (<i>atm</i>)) (Prim 'cmp (<i>atm</i> <i>atm</i>))</code>
<i>tail</i>	<code>::= (Goto <i>label</i>)</code> <code> (IfStmt (Prim <i>cmp</i> (<i>atm</i> <i>atm</i>)) (Goto <i>label</i>) (Goto <i>label</i>))</code>
<hr/>	
<i>atm</i>	<code>::= (Void)</code>
<i>stmt</i>	<code>::= (Prim 'read ())</code>
<hr/>	
<i>exp</i>	<code>::= (Allocate <i>int</i> <i>type</i>)</code> <code> (Prim vector-ref (<i>atm</i> (Int <i>int</i>)))</code> <code> (Prim vector-set! (<i>atm</i> (Int <i>int</i>) <i>atm</i>))</code> <code> (Prim vector-length (<i>atm</i>))</code> <code> (GlobalValue <i>var</i>)</code>
<i>stmt</i>	<code>::= (Prim vector-set! (<i>atm</i> (Int <i>int</i>) <i>atm</i>))</code> <code> (Collect <i>int</i>)</code>
<hr/>	
<i>exp</i>	<code>::= (FunRef <i>label</i> <i>int</i>) (Call <i>atm</i> (<i>atm</i> ...))</code>
<i>tail</i>	<code>::= (TailCall <i>atm</i> <i>atm</i> ...)</code>
<i>def</i>	<code>::= (Def <i>label</i> ([<i>var</i>:<i>type</i>] ...) <i>type</i> <i>info</i> ((<i>label</i> . <i>tail</i>) ...))</code>
<hr/>	
<i>exp</i>	<code>::= (AllocateClosure <i>int</i> <i>type</i> <i>int</i>)</code>
<i>op</i>	<code>::= procedure-arity</code>
<hr/>	
<i>exp</i>	<code>::= (Prim 'any-vector-ref (<i>atm</i> <i>atm</i>))</code> <code> (Prim 'any-vector-set! (list <i>atm</i> <i>atm</i> <i>atm</i>))</code> <code> (ValueOf <i>atm</i> <i>f</i><i>type</i>)</code>
<i>tail</i>	<code>::= (Exit)</code>
<i>C_{Any}</i>	<code>::= (ProgramDefs <i>info</i> (<i>def</i> ...))</code>

Figure 9.11

The abstract syntax of C_{Any} , extending C_{Clos} (Figure 8.8).

```
(Assign lhs (Prim 'make-any (list e (Int tag))))
⇒
movq e', lhs'
orq $tag, lhs'
```

tag-of-any Recall that the **tag-of-any** operation extracts the type tag from a value of type **Any**. The type tag is the bottom three bits, so we obtain the tag by taking the bitwise-and of the value with 111 (7 in decimal).

```
(Assign lhs (Prim 'tag-of-any (list e)))
⇒
movq e', lhs'
andq $7, lhs'
```

ValueOf The instructions for **ValueOf** also differ depending on whether the type T is a pointer (tuple or function) or not (integer or Boolean). The following shows

the instruction selection for integers and Booleans. We produce an untagged value by shifting it to the right by 3 bits.

```
(Assign lhs (ValueOf e T))
⇒
movq e', lhs'
sarq $3, lhs'
```

In the case for tuples and procedures, we zero-out the rightmost 3 bits. We accomplish this by creating the bit pattern ...0111 (7 in decimal) and apply bitwise-not to obtain ...11111000 (-8 in decimal) which we `movq` into the destination `lhs'`. Finally, we apply `andq` with the tagged value to get the desired result.

```
(Assign lhs (ValueOf e T))
⇒
movq $-8, lhs'
andq e', lhs'
```

any-vector-length The `any-vector-length` operation combines the effect of `ValueOf` with accessing the length of a tuple from the tag stored at the zero index of the tuple.

```
(Assign lhs (Prim 'any-vector-length (list e1)))
⇒
movq $-8, %r11
andq e'1, %r11
movq 0(%r11), %r11
andq $126, %r11
sarq $1, %r11
movq %r11, lhs'
```

any-vector-ref This operation combines the effect of `ValueOf` with reading an element of the tuple (see Section 6.7). However, the index may be an arbitrary atom so instead of computing the offset at compile time, we must generate instructions to compute the offset at runtime as follows. Note the use of the new instruction `imulq`.

```
(Assign lhs (Prim 'any-vector-ref (list e1 e2)))
⇒
movq -111, %r11
andq e'1, %r11
movq e'2, %rax
addq $1, %rax
imulq $8, %rax
addq %rax, %r11
movq 0(%r11) lhs'
```

any-vector-set! The code generation for `any-vector-set!` is analogous to the above translation for reading from a tuple.

9.9 Register Allocation for \mathcal{L}_{Any}

There is an interesting interaction between tagged values and garbage collection that has an impact on register allocation. A variable of type **Any** might refer to a tuple and therefore it might be a root that needs to be inspected and copied during garbage collection. Thus, we need to treat variables of type **Any** in a similar way to variables of tuple type for purposes of register allocation. In particular,

- If a variable of type **Any** is live during a function call, then it must be spilled. This can be accomplished by changing `build_interference` to mark all variables of type **Any** that are live after a `callq` as interfering with all the registers.
- If a variable of type **Any** is spilled, it must be spilled to the root stack instead of the normal procedure call stack.

Another concern regarding the root stack is that the garbage collector needs to differentiate between (1) plain old pointers to tuples, (2) a tagged value that points to a tuple, and (3) a tagged value that is not a tuple. We enable this differentiation by choosing not to use the tag `000` in the `tagof` function. Instead, that bit pattern is reserved for identifying plain old pointers to tuples. That way, if one of the first three bits is set, then we have a tagged value and inspecting the tag can differentiate between tuples (`010`) and the other kinds of values.

Exercise 37 Expand your compiler to handle \mathcal{L}_{Dyn} as outlined in this chapter. Create tests for \mathcal{L}_{Dyn} by adapting ten of your previous test programs by removing type annotations. Add 5 more tests programs that specifically rely on the language being dynamically typed. That is, they should not be legal programs in a statically typed language, but nevertheless, they should be valid \mathcal{L}_{Dyn} programs that run to completion without error.

Figure 9.12 provides an overview of the passes needed for the compilation of \mathcal{L}_{Dyn} .

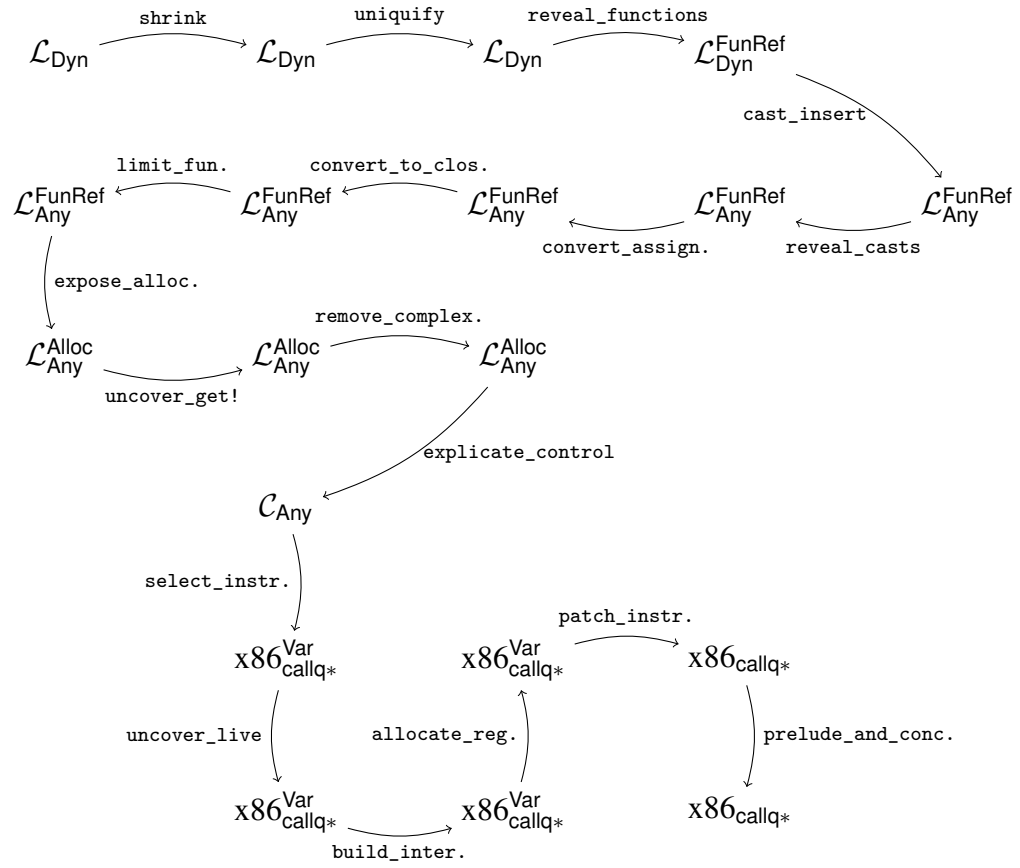
**Figure 9.12**

Diagram of the passes for \mathcal{L}_{Dyn} , a dynamically typed language.

10

Gradual Typing

This chapter studies a language, $\mathcal{L}_?$, in which the programmer can choose between static and dynamic type checking in different parts of a program, thereby mixing the statically typed \mathcal{L}_λ language with the dynamically typed \mathcal{L}_{Dyn} . There are several approaches to mixing static and dynamic typing, including multi-language integration (Tobin-Hochstadt and Felleisen 2006; Matthews and Findler 2007) and hybrid type checking (Flanagan 2006; Gronski et al. 2006). In this chapter we focus on *gradual typing*, in which the programmer controls the amount of static versus dynamic checking by adding or removing type annotations on parameters and variables (Anderson and Drossopoulou 2003; Siek and Taha 2006). The concrete syntax of $\mathcal{L}_?$ is defined in Figure 10.1 and its abstract syntax is defined in Figure 10.2. The main syntactic difference between \mathcal{L}_λ and $\mathcal{L}_?$ is the additional *param* and *ret* non-terminals that make type annotations optional. The return types are not optional in the abstract syntax; the parser fills in `Any` when the return type is not specified in the concrete syntax.

Both the type checker and the interpreter for $\mathcal{L}_?$ require some interesting changes to enable gradual typing, which we discuss in the next two sections in the context of the `map` example from Chapter 7. In Figure 10.3 we revisit the `map` example, omitting the type annotations from the `inc` function.

10.1 Type Checking $\mathcal{L}_?$ and $\mathcal{L}_{\text{Cast}}$

The type checker for $\mathcal{L}_?$ uses the `Any` type for missing parameter and return types. For example, the `x` parameter of `inc` in Figure 10.3 is given the type `Any` and the return type of `inc` is `Any`. Next consider the `+` operator inside `inc`. It expects both arguments to have type `Integer`, but its first argument `x` has type `Any`. In a gradually typed language, such differences are allowed so long as the types are *consistent*, that is, they are equal except in places where there is an `Any` type. The type `Any` is consistent with every other type. Figure 10.4 defines the `consistent?` predicate.

Returning to the `map` example of Figure 10.3, the `inc` function has type `(Any -> Any)` but parameter `f` of `map` has type `(Integer -> Integer)`. The type checker for $\mathcal{L}_?$ allows this because the two types are consistent. In particular, `->` is equal to `->` and because `Any` is consistent with `Integer`.

<i>type</i>	::=	Integer
<i>exp</i>	::=	<i>int</i> (read) (- <i>exp</i>) (+ <i>exp exp</i>) (- <i>exp exp</i>)
<i>exp</i>	::=	<i>var</i> (let ([<i>var exp</i>]) <i>exp</i>)
<i>type</i>	::=	Boolean
<i>bool</i>	::=	#t #f
<i>cmp</i>	::=	eq? < <= > >=
<i>exp</i>	::=	<i>bool</i> (and <i>exp exp</i>) (or <i>exp exp</i>) (not <i>exp</i>) (<i>cmp exp exp</i>) (if <i>exp exp exp</i>)
<i>type</i>	::=	Void
<i>exp</i>	::=	(set! <i>var exp</i>) (begin <i>exp* exp</i>) (while <i>exp exp</i>) (void)
<i>type</i>	::=	(Vector <i>type*</i>)
<i>exp</i>	::=	(vector <i>exp*</i>) (vector-length <i>exp</i>) (vector-ref <i>exp int</i>) (vector-set! <i>exp int exp</i>)
<i>type</i>	::=	(<i>type</i> ... -> <i>type</i>)
<i>param</i>	::=	<i>var</i> [<i>var: type</i>]
<i>ret</i>	::=	ϵ : <i>type</i>
<i>exp</i>	::=	(<i>exp exp</i> ...) (lambda: (<i>param</i> ...) <i>ret exp</i>) (procedure-arity <i>exp</i>)
<i>def</i>	::=	(define (<i>var param</i> ...) <i>ret exp</i>)
$\mathcal{L}_?$::=	<i>def</i> ... <i>exp</i>

Figure 10.1

The concrete syntax of $\mathcal{L}_?$, extending \mathcal{L}_{Top} (Figure 6.1).

<i>type</i>	::=	Integer
<i>op</i>	::=	read + -
<i>exp</i>	::=	(Int <i>int</i>) (Prim <i>op</i> (<i>exp</i> ...))
<i>exp</i>	::=	(Var <i>var</i>) (Let <i>var exp exp</i>)
<i>type</i>	::=	Boolean
<i>bool</i>	::=	#t #f
<i>cmp</i>	::=	eq? < <= > >=
<i>op</i>	::=	<i>cmp</i> and or not
<i>exp</i>	::=	(Bool <i>bool</i>) (If <i>exp exp exp</i>)
<i>type</i>	::=	Void
<i>exp</i>	::=	(SetBang <i>var exp</i>) (Begin <i>exp* exp</i>) (WhileLoop <i>exp exp</i>) (Void)
<i>type</i>	::=	(Vector <i>type*</i>)
<i>op</i>	::=	vector vector-length
<i>exp</i>	::=	(Prim vector-ref (<i>exp</i> (Int <i>int</i>))) (Prim vector-set! (<i>exp</i> (Int <i>int</i>) <i>exp</i>)) (HasType <i>exp type</i>)
<i>type</i>	::=	(<i>type</i> ... -> <i>type</i>)
<i>param</i>	::=	<i>var</i> [<i>var: type</i>]
<i>exp</i>	::=	(Apply <i>exp exp</i> ...) (Lambda (<i>param</i> ...) <i>type exp</i>)
<i>op</i>	::=	procedure-arity
<i>def</i>	::=	(Def <i>var</i> (<i>param</i> ...) <i>type</i> '() <i>exp</i>)
$\mathcal{L}_?$::=	(ProgramDefsExp '() (<i>def</i> ...) <i>exp</i>)

Figure 10.2

The abstract syntax of $\mathcal{L}_?$, extending \mathcal{L}_{Top} (Figure 6.2).

```

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

(define (inc x) (+ x 1))

(vector-ref (map inc (vector 0 41)) 1)

```

Figure 10.3

A partially-typed version of the `map` example.

```

(define/public (consistent? t1 t2)
  (match* (t1 t2)
    [('Integer 'Integer) #t]
    [('Boolean 'Boolean) #t]
    [('Void 'Void) #t]
    [('Any t2) #t]
    [(t1 'Any) #t]
    [(^(Vector ,ts1 ...) ^ (Vector ,ts2 ...))
     (for/and ([t1 ts1] [t2 ts2]) (consistent? t1 t2))]
    [(^(,ts1 ... -> ,rt1) ^ (,ts2 ... -> ,rt2))
     (and (for/and ([t1 ts1] [t2 ts2]) (consistent? t1 t2))
          (consistent? rt1 rt2))]
    [(other wise) #f]))

```

Figure 10.4

The consistency predicate on types.

Next consider a program with an error, such as applying `map` to a function that sometimes returns a `Boolean`, as shown in Figure 10.5. The type checker for $\mathcal{L}_?$ accepts this program because the type of `maybe-inc` is consistent with the type of parameter `f` of `map`, that is, $(\text{Any} \rightarrow \text{Any})$ is consistent with $(\text{Integer} \rightarrow \text{Integer})$. One might say that a gradual type checker is optimistic in that it accepts programs that might execute without a runtime type error. Unfortunately, running this program with input `1` triggers an error when the `maybe-inc` function returns `#t`. The $\mathcal{L}_?$ language performs checking at runtime to ensure the integrity of the static types, such as the $(\text{Integer} \rightarrow \text{Integer})$ annotation on parameter `f` of `map`. This runtime checking is carried out by a new `Cast` form that is inserted by the type checker. Thus, the output of the type checker is a program in the $\mathcal{L}_{\text{Cast}}$ language, which adds `Cast` and `Any` to \mathcal{L}_λ .

Figure 10.6 shows the output of the type checker for `map` and `maybe-inc`. The idea is that `Cast` is inserted every time the type checker sees two types that are consistent but not equal. In the `inc` function, `x` is cast to `Integer` and the result of the `+` is cast to `Any`. In the call to `map`, the `inc` argument is cast from $(\text{Any} \rightarrow \text{Any})$ to $(\text{Integer} \rightarrow \text{Integer})$.

The type checker for $\mathcal{L}_?$ is defined in Figures 10.7, 10.8, and 10.9.

```

(define (map [f : (Integer -> Integer)]
  [v : (Vector Integer Integer)])
  : (Vector Integer Integer)
  (vector (f (vector-ref v 0)) (f (vector-ref v 1))))
(define (inc x) (+ x 1))
(define (true) #t)
(define (maybe-inc x) (if (eq? 0 (read)) (inc x) (true)))

(vector-ref (map maybe-inc (vector 0 41)) 0)

```

Figure 10.5

A variant of the map example with an error.

```

(define (map [f : (Integer -> Integer)] [v : (Vector Integer Integer)])
  : (Vector Integer Integer)
  (vector (f (vector-ref v 0)) (f (vector-ref v 1))))
(define (inc [x : Any]) : Any
  (cast (+ (cast x Any Integer) 1) Integer Any))
(define (true) : Any (cast #t Boolean Any))
(define (maybe-inc [x : Any]) : Any
  (if (eq? 0 (read)) (inc x) (true)))

(vector-ref (map (cast maybe-inc (Any -> Any) (Integer -> Integer))
  (vector 0 41)) 0)

```

Figure 10.6

Output of type checking map and maybe-inc.

```

(define type-check-gradual-class
  (class type-check-Lambda-class
    (super-new)
    (inherit operator-types type-predicates)

    (define/override (type-check-exp env)
      (lambda (e)
        (define recur (type-check-exp env))
        (match e
          [(Prim 'vector-length (list e1))
           (define-values (e1~ t) (recur e1))
           (match t
             [(Vector ,ts ...)
              (values (Prim 'vector-length (list e1~)) 'Integer)]
             ['Any (values (Prim 'any-vector-length (list e1~)) 'Integer)]])]
          [(Prim 'vector-ref (list e1 e2))
           (define-values (e1~ t1) (recur e1))
           (define-values (e2~ t2) (recur e2))
           (check-consistent? t2 'Integer e)
           (match t1
             [(Vector ,ts ...)
              (match e2~
                [(Int i)
                 (unless (and (0 . <= . i) (i . < . (length ts)))
                   (error 'type-check "invalid index ~a in ~a" i e))
                 (values (Prim 'vector-ref (list e1~ (Int i))) (list-ref ts i))]
                [else (define e1^^ (make-cast e1~ t1 'Any))
                     (define e2^^ (make-cast e2~ t2 'Integer))
                     (values (Prim 'any-vector-ref (list e1^^ e2^^)) 'Any)]])]
             ['Any
              (define e2^^ (make-cast e2~ t2 'Integer))
              (values (Prim 'any-vector-ref (list e1~ e2^^)) 'Any)]
             [else (error 'type-check "expected vector not ~a\nin ~v" t1 e)]])]
          [(Prim 'vector-set! (list e1 e2 e3) )
           (define-values (e1~ t1) (recur e1))
           (define-values (e2~ t2) (recur e2))
           (define-values (e3~ t3) (recur e3))
           (check-consistent? t2 'Integer e)
           (match t1
             [(Vector ,ts ...)
              (match e2~
                [(Int i)
                 (unless (and (0 . <= . i) (i . < . (length ts)))
                   (error 'type-check "invalid index ~a in ~a" i e))
                 (check-consistent? (list-ref ts i) t3 e)
                 (define e3^^ (make-cast e3~ t3 (list-ref ts i)))
                 (values (Prim 'vector-set! (list e1~ (Int i) e3^^)) 'Void)]
                [else
                 (define e1^^ (make-cast e1~ t1 'Any))
                 (define e2^^ (make-cast e2~ t2 'Integer))
                 (define e3^^ (make-cast e3~ t3 'Any))
                 (values (Prim 'any-vector-set! (list e1^^ e2^^ e3^^)) 'Void)]])]
             ['Any
              (define e2^^ (make-cast e2~ t2 'Integer))
              (define e3^^ (make-cast e3~ t3 'Any))
              (values (Prim 'any-vector-set! (list e1~ e2^^ e3^^)) 'Void)]
             [else (error 'type-check "expected vector not ~a\nin ~v" t1 e)]])]
          [else (error 'type-check "expected vector not ~a\nin ~v" t1 e)]])

```

Figure 10.7

Type checker for the $\mathcal{L}_?$ language, part 1.

```

[(Prim 'eq? (list e1 e2))
 (define-values (e1~ t1) (recur e1))
 (define-values (e2~ t2) (recur e2))
 (check-consistent? t1 t2 e)
 (define T (meet t1 t2))
 (values (Prim 'eq? (list (make-cast e1~ t1 T) (make-cast e2~ t2 T)))
  'Boolean)]
[(Prim 'not (list e1))
 (define-values (e1~ t1) (recur e1))
 (match t1
  ['Any
   (recur (If (Prim 'eq? (list e1 (Inject (Bool #f) 'Boolean)))
    (Bool #t) (Bool #f)))]
  [else
   (define-values (t-ret new-es~)
    (type-check-op 'not (list t1) (list e1~) e))
   (values (Prim 'not new-es~) t-ret)]])
[(Prim 'and (list e1 e2))
 (recur (If e1 e2 (Bool #f)))]
[(Prim 'or (list e1 e2))
 (define tmp (gensym 'tmp))
 (recur (Let tmp e1 (If (Var tmp) (Var tmp) e2)))]
[(Prim op es)
 #:when (not (set-member? explicit-prim-ops op))
 (define-values (new-es ts)
  (for/lists (exprs types) ([e es]
   (recur e)))
 (define-values (t-ret new-es~) (type-check-op op ts new-es e))
 (values (Prim op new-es~) t-ret)]
[(If e1 e2 e3)
 (define-values (e1~ T1) (recur e1))
 (define-values (e2~ T2) (recur e2))
 (define-values (e3~ T3) (recur e3))
 (check-consistent? T2 T3 e)
 (match T1
  ['Boolean
   (define Tif (join T2 T3))
   (values (If e1~ (make-cast e2~ T2 Tif)
    (make-cast e3~ T3 Tif)) Tif)]
  ['Any
   (define Tif (meet T2 T3))
   (values (If (Prim 'eq? (list e1~ (Inject (Bool #f) 'Boolean)))
    (make-cast e3~ T3 Tif) (make-cast e2~ T2 Tif))
    Tif)]
  [else (error 'type-check "expected Boolean not ~a\nin ~v" T1 e)]]]
[(HasType e1 T)
 (define-values (e1~ T1) (recur e1))
 (check-consistent? T1 T)
 (values (make-cast e1~ T1 T) T)]
[(SetBang x e1)
 (define-values (e1~ T1) (recur e1))
 (define varT (dict-ref env x))
 (check-consistent? T1 varT e)
 (values (SetBang x (make-cast e1~ T1 varT)) 'Void)]
[(WhileLoop e1 e2)
 (define-values (e1~ T1) (recur e1))
 (check-consistent? T1 'Boolean e)
 (define-values (e2~ T2) ((type-check-exp env) e2))
 (values (WhileLoop (make-cast e1~ T1 'Boolean) e2~) 'Void)]

```

Figure 10.8

Type checker for the $\mathcal{L}_?$ language, part 2.

```

[(Apply e1 e2s)
 (define-values (e1~ T1) (recur e1))
 (define-values (e2s~ T2s) (for/lists (e* ty*) ([e2 e2s]) (recur e2)))
 (match T1
  [^(,T1ps ... -> ,T1rt)
   (for ([T2 T2s] [Tp T1ps])
    (check-consistent? T2 Tp e))
   (define e2s~ (for/list ([e2 e2s~] [src T2s] [tgt T1ps])
    (make-cast e2 src tgt)))
   (values (Apply e1~ e2s~) T1rt)]
  [^Any
   (define e1~ (make-cast e1~ 'Any
    ^(@(for/list ([e e2s]) 'Any) -> Any)))
   (define e2s~ (for/list ([e2 e2s~] [src T2s])
    (make-cast e2 src 'Any)))
   (values (Apply e1~ e2s~) 'Any)]
  [else (error 'type-check "expected function not ~a\nin ~v" T1 e)]])
[(Lambda params Tr e1)
 (define-values (xs Ts) (for/lists (l1 l2) ([p params])
  (match p
   [^[,x : ,T] (values x T)]
   [(? symbol? x) (values x 'Any)])))
 (define-values (e1~ T1)
  ((type-check-exp (append (map cons xs Ts) env)) e1))
 (check-consistent? Tr T1 e)
 (values (Lambda (for/list ([x xs] [T Ts]) ^[,x : ,T]) Tr
  (make-cast e1~ T1 Tr)) ^(@(Ts -> ,Tr)))
 [else ((super type-check-exp env) e)]
 )))

```

Figure 10.9Type checker for the $\mathcal{L}_?$ language, part 3.

```

(define/public (join t1 t2)
  (match* (t1 t2)
    [(('Integer 'Integer) 'Integer]
    [(('Boolean 'Boolean) 'Boolean]
    [(('Void 'Void) 'Void]
    [(('Any t2) t2]
    [(t1 'Any) t1]
    [(('Vector ,ts1 ...) `(Vector ,ts2 ...))
     `(Vector ,@(for/list ([t1 ts1] [t2 ts2]) (join t1 t2)))]
    [((' ,ts1 ... -> ,rt1) `( ,ts2 ... -> ,rt2))
     `( ,@(for/list ([t1 ts1] [t2 ts2]) (join t1 t2))
       -> ,(join rt1 rt2)))]))

(define/public (meet t1 t2)
  (match* (t1 t2)
    [(('Integer 'Integer) 'Integer]
    [(('Boolean 'Boolean) 'Boolean]
    [(('Void 'Void) 'Void]
    [(('Any t2) 'Any]
    [(t1 'Any) 'Any]
    [(('Vector ,ts1 ...) `(Vector ,ts2 ...))
     `(Vector ,@(for/list ([t1 ts1] [t2 ts2]) (meet t1 t2)))]
    [((' ,ts1 ... -> ,rt1) `( ,ts2 ... -> ,rt2))
     `( ,@(for/list ([t1 ts1] [t2 ts2]) (meet t1 t2))
       -> ,(meet rt1 rt2)))]))

(define/public (make-cast e src tgt)
  (cond [(equal? src tgt) e] [else (Cast e src tgt)]))

(define/public (check-consistent? t1 t2 e)
  (unless (consistent? t1 t2)
    (error 'type-check "~a is inconsistent with ~a\nin ~v" t1 t2 e)))

(define/override (type-check-op op arg-types args e)
  (match (dict-ref (operator-types) op)
    [( ,param-types . ,return-type)
     (for ([at arg-types] [pt param-types])
       (check-consistent? at pt e))
     (values return-type
              (for/list ([e args] [s arg-types] [t param-types])
                (make-cast e s t))))]
    [else (error 'type-check-op "unrecognized ~a" op)]))

(define explicit-prim-ops
  (set-union
    (type-predicates)
    (set 'procedure-arity 'eq?
          'vector 'vector-length 'vector-ref 'vector-set!
          'any-vector-length 'any-vector-ref 'any-vector-set!)))

(define/override (fun-def-type d)
  (match d
    [(Def f params rt info body)
     (define ps
       (for/list ([p params])
         (match p
           [( ,x : ,T) T]
           [(? symbol?) 'Any]
           [else (error 'fun-def-type "unmatched parameter ~a" p)])))
     `( ,@ps -> ,rt)]
    [else (error 'fun-def-type "ill-formed function definition in ~a" d)]))

```

Figure 10.10

Auxiliary functions for type checking $\mathcal{L}_?$.


```

(define (map! [f : (Any -> Any)]
             [v : (Vector Any Any)]) : Void
  (begin
    (vector-set! v 0 (f (vector-ref v 0)))
    (vector-set! v 1 (f (vector-ref v 1))))

(define (inc x) (+ x 1))

(let ([v (vector 0 41)])
  (begin (map! inc v) (vector-ref v 1)))

```

Figure 10.11

An example involving casts on vectors.

10.2 Interpreting $\mathcal{L}_{\text{Cast}}$

The runtime behavior of first-order casts is straightforward, that is, casts involving simple types such as `Integer` and `Boolean`. For example, a cast from `Integer` to `Any` can be accomplished with the `Inject` operator of \mathcal{L}_{Any} , which puts the integer into a tagged value (Figure 9.8). Similarly, a cast from `Any` to `Integer` is accomplished with the `Project` operator, that is, by checking the value’s tag and either retrieving the underlying integer or signaling an error if it the tag is not the one for integers (Figure 9.9). Things get more interesting for higher-order casts, that is, casts involving function or tuple types.

Consider the cast of the function `maybe-inc` from `(Any -> Any)` to `(Integer -> Integer)`. When a function flows through this cast at runtime, we can’t know in general whether the function will always return an integer.¹⁵ The $\mathcal{L}_{\text{Cast}}$ interpreter therefore delays the checking of the cast until the function is applied. This is accomplished by wrapping `maybe-inc` in a new function that casts its parameter from `Integer` to `Any`, applies `maybe-inc`, and then casts the return value from `Any` to `Integer`.

Turning our attention to casts involving tuple types, we consider the example in Figure 10.11 that defines a partially-typed version of `map` whose parameter `v` has type `(Vector Any Any)` and that updates `v` in place instead of returning a new tuple. So we name this function `map!`. We apply `map!` to a tuple of integers, so the type checker inserts a cast from `(Vector Integer Integer)` to `(Vector Any Any)`. A naive way for the $\mathcal{L}_{\text{Cast}}$ interpreter to cast between tuple types would be to build a new tuple whose elements are the result of casting each of the original elements to the appropriate target type. However, this approach is only valid for immutable tuples; and our tuples are mutable. In the example of Figure 10.11, if the cast created a new tuple, then the updates inside of `map!` would happen to the new tuple and not the original one.

15. Predicting the return value of a function is equivalent to the halting problem, which is undecidable.

```

(define (map! [f : (Any -> Any)] v) : Void
  (begin
    (vector-set! v 0 (f (vector-ref v 0)))
    (vector-set! v 1 (f (vector-ref v 1)))))

(define (inc x) (+ x 1))

(let ([v (vector 0 41)])
  (begin (map! inc v) (vector-ref v 1)))

```

Figure 10.12Casting a tuple to *Any*.

Instead the interpreter needs to create a new kind of value, a *tuple proxy*, that intercepts every tuple operation. On a read, the proxy reads from the underlying tuple and then applies a cast to the resulting value. On a write, the proxy casts the argument value and then performs the write to the underlying tuple. For the first `(vector-ref v 0)` in `map!`, the proxy casts 0 from `Integer` to `Any`. For the first `vector-set!`, the proxy casts a tagged 1 from `Any` to `Integer`.

The final category of cast that we need to consider are casts between the `Any` type and either a function or a tuple type. Figure 10.12 shows a variant of `map!` in which parameter `v` does not have a type annotation, so it is given type `Any`. In the call to `map!`, the tuple has type `(Vector Integer Integer)` so the type checker inserts a cast from `(Vector Integer Integer)` to `Any`. A first thought is to use `Inject`, but that doesn't work because `(Vector Integer Integer)` is not a flat type. Instead, we must first cast to `(Vector Any Any)` (which is flat) and then inject to `Any`.

The $\mathcal{L}_{\text{Cast}}$ interpreter uses an auxiliary function named `apply-cast` to cast a value from a source type to a target type, shown in Figure 10.13. You'll find that it handles all of the kinds of casts that we've discussed in this section.

The interpreter for $\mathcal{L}_{\text{Cast}}$ is defined in Figure 10.14, with the case for `Cast` dispatching to `apply-cast`. To handle the addition of tuple proxies, we update the tuple primitives in `interp-op` using the functions in Figure 10.15.

10.3 Lower Casts

The next step in the journey towards x86 is the `lower-casts` pass that translates the casts in $\mathcal{L}_{\text{Cast}}$ to the lower-level `Inject` and `Project` operators and a new operator for creating tuple proxies, extending the \mathcal{L}_λ language to create $\mathcal{L}_{\text{Proxy}}$. We recommend creating an auxiliary function named `lower-cast` that takes an expression (in $\mathcal{L}_{\text{Cast}}$), a source type, and a target type, and translates it to expression in $\mathcal{L}_{\text{Proxy}}$ that has the same behavior as casting the expression from the source to the target type in the interpreter.

The `lower-cast` function can follow a code structure similar to the `apply-cast` function (Figure 10.13) used in the interpreter for $\mathcal{L}_{\text{Cast}}$ because it must handle the same cases as `apply-cast` and it needs to mimic the behavior of `apply-cast`. The

```

(define/public (apply-cast v s t)
  (match* (s t)
    [(t1 t2) #:when (equal? t1 t2) v]
    [('Any t2)
     (match t2
       [(,ts ... -> ,rt)
        (define any->any `(,@(for/list ([t ts]) 'Any) -> Any))
        (define v~ (apply-project v any->any))
        (apply-cast v~ any->any `(,@ts -> ,rt)))]
       [(Vector ,ts ...)
        (define vec-any `(Vector ,@(for/list ([t ts]) 'Any))
        (define v~ (apply-project v vec-any))
        (apply-cast v~ vec-any `(Vector ,@ts))]
       [else (apply-project v t2)]]])
    [(t1 'Any)
     (match t1
       [(,ts ... -> ,rt)
        (define any->any `(,@(for/list ([t ts]) 'Any) -> Any))
        (define v~ (apply-cast v `(,@ts -> ,rt) any->any))
        (apply-inject v~ (any-tag any->any))]
       [(Vector ,ts ...)
        (define vec-any `(Vector ,@(for/list ([t ts]) 'Any))
        (define v~ (apply-cast v `(Vector ,@ts) vec-any))
        (apply-inject v~ (any-tag vec-any))]
       [else (apply-inject v (any-tag t1) )])])
    [(Vector ,ts1 ...) `(Vector ,ts2 ...))
    (define x (gensym 'x))
    (define cast-reads (for/list ([t1 ts1] [t2 ts2])
                               `(function (,x) ,(Cast (Var x) t1 t2) ())))
    (define cast-writes
      (for/list ([t1 ts1] [t2 ts2])
        `(function (,x) ,(Cast (Var x) t2 t1) ())))
    `(vector-proxy ,(vector v (apply vector cast-reads)
                             (apply vector cast-writes)))
    [(,ts1 ... -> ,rt1) `(,ts2 ... -> ,rt2))
    (define xs (for/list ([t2 ts2]) (gensym 'x)))
    `(function ,xs ,(Cast
      (Apply (Value v)
        (for/list ([x xs] [t1 ts1] [t2 ts2])
          (Cast (Var x) t2 t1)))
      rt1 rt2) ()))
  ))

```

Figure 10.13

The `apply-cast` auxiliary method.

most interesting cases are those concerning the casts between two tuple types and between two function types.

As mentioned in Section 10.2, a cast from one tuple type to another tuple type is accomplished by creating a proxy that intercepts the operations on the underlying tuple. Here we make the creation of the proxy explicit with the `vector-proxy` primitive operation. It takes three arguments, the first is an expression for the tuple, the second is a tuple of functions for casting an element that is being read from the tuple, and the third is a tuple of functions for casting an element that is being written to the tuple. You can create the functions using `Lambda`. Also, as we

```

(define interp-Lcast-class
  (class interp-Llambda-class
    (super-new)
    (inherit apply-fun apply-inject apply-project)

    (define/override (interp-op op)
      (match op
        ['vector-length guarded-vector-length]
        ['vector-ref guarded-vector-ref]
        ['vector-set! guarded-vector-set!]
        ['any-vector-ref (lambda (v i)
                           (match v [`(tagged ,v^ ,tg)
                                     (guarded-vector-ref v^ i)]))]
        ['any-vector-set! (lambda (v i a)
                            (match v [`(tagged ,v^ ,tg)
                                      (guarded-vector-set! v^ i a)]))]
        ['any-vector-length (lambda (v)
                              (match v [`(tagged ,v^ ,tg)
                                        (guarded-vector-length v^)]))]
        [else (super interp-op op)]
      ))

    (define/override ((interp-exp env) e)
      (define (recur e) ((interp-exp env) e))
      (match e
        [(Value v) v]
        [(Cast e src tgt) (apply-cast (recur e) src tgt)]
        [else ((super interp-exp env) e)]]
      ))

    (define (interp-Lcast p)
      (send (new interp-Lcast-class) interp-program p))

```

Figure 10.14

The interpreter for $\mathcal{L}_{\text{Cast}}$.

shall see in the next section, we need to differentiate these tuples from the user-created ones, so we recommend using a new primitive operator named **raw-vector** instead of **vector** to create these tuples of functions. Figure 10.16 shows the output of **lower-casts** on the example in Figure 10.11 that involved casting a tuple of integers to a tuple of **Any**.

A cast from one function type to another function type is accomplished by generating a **Lambda** whose parameter and return types match the target function type. The body of the **Lambda** should cast the parameters from the target type to the source type. (Yes, backwards! Functions are contravariant in the parameters.). Afterwards, call the underlying function and then cast the result from the source return type to the target return type. Figure 10.17 shows the output of the **lower-casts** pass on the **map** example in Figure 10.3. Note that the **inc** argument in the call to **map** is wrapped in a **lambda**.

```

(define (guarded-vector-ref vec i)
  (match vec
    [(vector-proxy ,proxy)
     (define val (guarded-vector-ref (vector-ref proxy 0) i))
     (define rd (vector-ref (vector-ref proxy 1) i))
     (apply-fun rd (list val) 'guarded-vector-ref)]
    [else (vector-ref vec i)]))

(define (guarded-vector-set! vec i arg)
  (match vec
    [(vector-proxy ,proxy)
     (define wr (vector-ref (vector-ref proxy 2) i))
     (define arg^ (apply-fun wr (list arg) 'guarded-vector-set!))
     (guarded-vector-set! (vector-ref proxy 0) i arg^)]
    [else (vector-set! vec i arg)]))

(define (guarded-vector-length vec)
  (match vec
    [(vector-proxy ,proxy)
     (guarded-vector-length (vector-ref proxy 0))]
    [else (vector-length vec)]))

```

Figure 10.15

The guarded-vector auxiliary functions.

```

(define (map! [f : (Any -> Any)] [v : (Vector Any Any)]) : Void
  (begin
    (vector-set! v 0 (f (vector-ref v 0)))
    (vector-set! v 1 (f (vector-ref v 1)))))

(define (inc [x : Any]) : Any
  (inject (+ (project x Integer) 1) Integer))

(let ([v (vector 0 41)])
  (begin
    (map! inc (vector-proxy v
      (raw-vector (lambda: ([x9 : Integer]) : Any
        (inject x9 Integer))
        (lambda: ([x9 : Integer]) : Any
        (inject x9 Integer)))
      (raw-vector (lambda: ([x9 : Any]) : Integer
        (project x9 Integer))
        (lambda: ([x9 : Any]) : Integer
        (project x9 Integer)))))
    (vector-ref v 1)))

```

Figure 10.16

Output of lower-casts on the example in Figure 10.11.

10.4 Differentiate Proxies

So far the job of differentiating tuples and tuple proxies has been the job of the interpreter. For example, the interpreter for $\mathcal{L}_{\text{Cast}}$ implements `vector-ref` using

```

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

(define (inc [x : Any]) : Any
  (inject (+ (project x Integer) 1) Integer))

(vector-ref (map (lambda: ([x9 : Integer]) : Integer
  (project (inc (inject x9 Integer)) Integer))
  (vector 0 41)) 1)

```

Figure 10.17

Output of `lower-casts` on the example in Figure 10.3.

the `guarded-vector-ref` function in Figure 10.15. In the `differentiate-proxies` pass we shift this responsibility to the generated code.

We begin by designing the output language $\mathcal{L}_{\text{PVec}}$. In $\mathcal{L}_?$ we used the type `Vector` for both real tuples and tuple proxies. In $\mathcal{L}_{\text{PVec}}$ we return the `Vector` type to its original meaning, as the type of real tuples, and we introduce a new type, `PVector`, whose values can be either real tuples or tuple proxies. This new type comes with a suite of new primitive operations for creating and using values of type `PVector`. A proxy is represented by a tuple containing three things: 1) the underlying tuple, 2) a tuple of functions for casting elements that are read from the tuple, and 3) a tuple of functions for casting values to be written to the tuple. So we define the following abbreviation for the type of a tuple proxy:

$$\text{Proxy}(T \dots \Rightarrow T' \dots) = (\text{Vector} (\text{PVector } T \dots) R W) \rightarrow (\text{PVector } T' \dots)$$

where $R = (\text{Vector} (T \rightarrow T') \dots)$ and $W = (\text{Vector} (T' \rightarrow T) \dots)$. Next we describe each of the new primitive operations.

inject-vector : (Vector T ...) → (PVector T ...)

This operation brands a vector as a value of the `PVector` type.

inject-proxy : Proxy(T ... ⇒ T' ...) → (PVector T' ...)

This operation brands a vector proxy as value of the `PVector` type.

proxy? : (PVector T ...) → Boolean

This returns true if the value is a tuple proxy and false if it is a real tuple.

project-vector : (PVector T ...) → (Vector T ...)

Assuming that the input is a tuple, this operation returns the tuple.

proxy-vector-length : (PVector T ...) → Boolean

Given a tuple proxy, this operation returns the length of the tuple.

proxy-vector-ref : (PVector T ...) → (i : Integer) → T_i

Given a tuple proxy, this operation returns the *i*th element of the tuple.

proxy-vector-set! : (PVector T ...) → (i : Integer) → T_i → Void

Given a tuple proxy, this operation writes a value to the *i*th element of the tuple.

Now to discuss the translation that differentiates tuples from proxies. First, every type annotation in the program is translated (recursively) to replace **Vector** with **PVector**. Next, we insert uses of **PVector** operations in the appropriate places. For example, we wrap every tuple creation with an **inject-vector**.

```
(vector e1 ... en)
⇒
(inject-vector (vector e'1 ... e'n))
```

The **raw-vector** operator that we introduced in the previous section does not get injected.

```
(raw-vector e1 ... en)
⇒
(vector e'1 ... e'n)
```

The **vector-proxy** primitive translates as follows.

```
(vector-proxy e1 e2 e3)
⇒
(inject-proxy (vector e'1 e'2 e'3))
```

We translate the tuple operations into conditional expressions that check whether the value is a proxy and then dispatch to either the appropriate proxy tuple operation or the regular tuple operation. For example, the following is the translation for **vector-ref**.

```
(vector-ref e1 i)
⇒
(let ([v e1])
  (if (proxy? v)
      (proxy-vector-ref v i)
      (vector-ref (project-vector v) i)))
```

Note in the case of a real tuple, we must apply **project-vector** before the **vector-ref**.

10.5 Reveal Casts

Recall that the **reveal-casts** pass (Section 9.5) is responsible for lowering **Inject** and **Project** into lower-level operations. In particular, **Project** turns into a conditional expression that inspects the tag and retrieves the underlying value. Here we need to augment the translation of **Project** to handle the situation when the target type is **PVector**. Instead of using **vector-length** we need to use **proxy-vector-length**.

```
(project e (PVector Any1 ... Anyn))
⇒
(let tmp e'
  (if (eq? (tag-of-any tmp 2))
      (let tup (value-of tmp (PVector Any ... Any))
        (if (eq? (proxy-vector-length tup) n) tup (exit)))
      (exit))))
```

10.6 Closure Conversion

The closure conversion pass only requires one minor adjustment. The auxiliary function that translates type annotations needs to be updated to handle the `PVector` type.

10.7 Explicate Control

Update the `explicate_control` pass to handle the new primitive operations on the `PVector` type.

10.8 Select Instructions

Recall that the `select_instructions` pass is responsible for lowering the primitive operations into x86 instructions. So we need to translate the new `PVector` operations to x86. To do so, the first question we need to answer is how to differentiate the two kinds of values (tuples and proxies) that can inhabit `PVector`. We need just one bit to accomplish this, and use the bit in position 57 of the 64-bit tag at the front of every tuple (see Figure 6.8). So far, this bit has been set to 0, so for `inject-vector` we leave it that way.

```
(Assign lhs (Prim 'inject-vector (list e1)))
⇒
movq e1, lhs'
```

On the other hand, `inject-proxy` sets bit 57 to 1.

```
(Assign lhs (Prim 'inject-proxy (list e1)))
⇒
movq e1, %r11
movq (1<<57), %rax
orq 0(%r11), %rax
movq %rax, 0(%r11)
movq %r11, lhs'
```

The `proxy?` operation consumes the information so carefully stashed away by `inject-vector` and `inject-proxy`. It isolates the 57th bit to tell whether the value is a real tuple or a proxy.

```
(Assign lhs (Prim 'proxy? (list e)))
⇒
movq e1, %r11
movq 0(%r11), %rax
sarq $57, %rax
andq $1, %rax
movq %rax, lhs'
```

The `project-vector` operation is straightforward to translate, so we leave it up to the reader.

Regarding the `proxy-vector` operations, the runtime provides procedures that implement them (they are recursive functions!) so here we simply need to translate these tuple operations into the appropriate function call. For example, here is the translation for `proxy-vector-ref`.

```
(Assign lhs (Prim 'proxy-vector-ref (list e1 e2)))
⇒
movq e'1, %rdi
movq e'2, %rsi
callq proxy_vector_ref
movq %rax, lhs'
```

We have another batch of tuple operations to deal with, those for the `Any` type. Recall that the type checker for $\mathcal{L}_?$ generates an `any-vector-ref` when there is a `vector-ref` on something of type `Any`, and similarly for `any-vector-set!` and `any-vector-length` (Figure 10.7). In Section 9.8 we selected instructions for these operations based on the idea that the underlying value was a real tuple. But in the current setting, the underlying value is of type `PVector`. So `any-vector-ref` can be translated follows. We begin by projecting the underlying value out of the tagged value and then call the `proxy_vector_ref` procedure in the runtime.

```
(Assign lhs (Prim 'any-vector-ref (list e1 e2)))
movq -111, %rdi
andq e'1, %rdi
movq e'2, %rsi
callq proxy_vector_ref
movq %rax, lhs'
```

The `any-vector-set!` and `any-vector-length` operators can be translated in a similar way.

Exercise 38 Implement a compiler for the gradually-typed $\mathcal{L}_?$ language by extending and adapting your compiler for \mathcal{L}_λ . Create 10 new partially-typed test programs. In addition to testing with these new programs, also test your compiler on all the tests for \mathcal{L}_λ and tests for \mathcal{L}_{Dyn} . Sometimes you may get a type checking error on the \mathcal{L}_{Dyn} programs but you can adapt them by inserting a cast to the `Any` type around each subexpression causing a type error. While \mathcal{L}_{Dyn} does not have explicit casts, you can induce one by wrapping the subexpression `e` with a call to an un-annotated identity function, like this: `((lambda (x) x) e)`.

Figure 10.18 provides an overview of the passes needed for the compilation of $\mathcal{L}_?$.

10.9 Further Reading

This chapter just scratches the surface of gradual typing. The basic approach described here is missing two key ingredients that one would want in a implementation of gradual typing: blame tracking (Tobin-Hochstadt and Felleisen 2006; Wadler and Findler 2009) and space-efficient casts (Herman, Tomb, and Flanagan 2007, 2010). The problem addressed by blame tracking is that when a cast

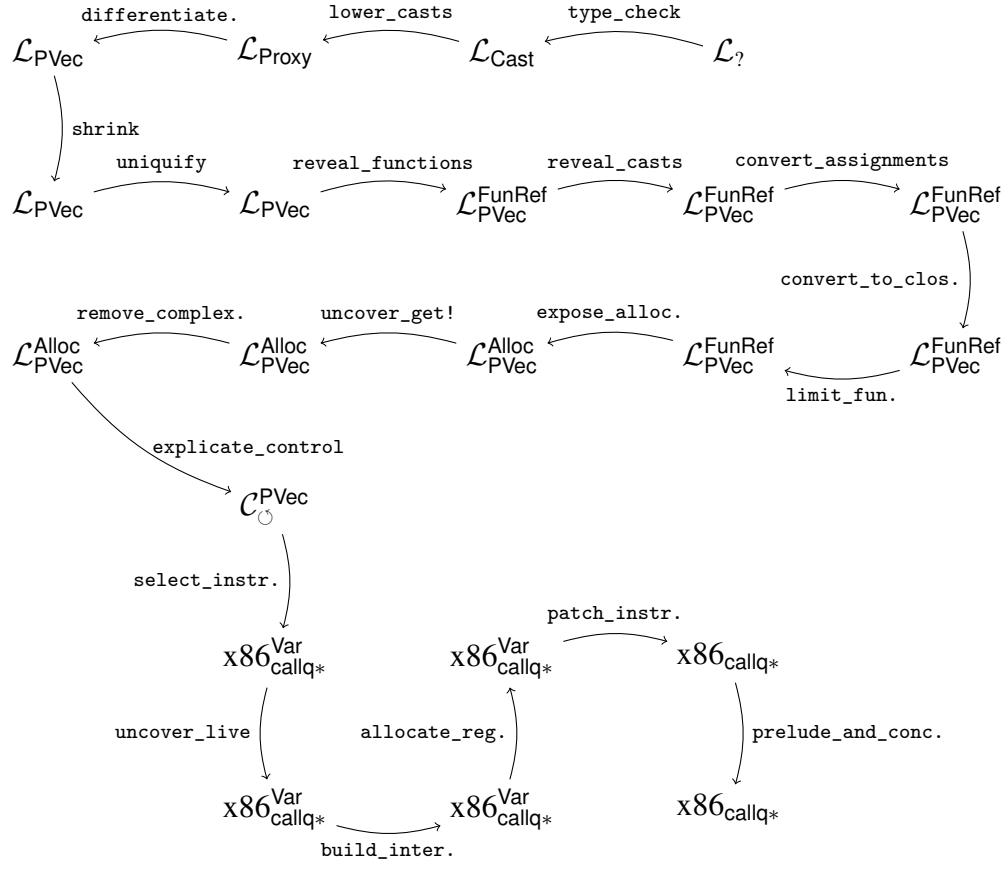
on a higher-order value fails, it often does so at a point in the program that is far removed from the original cast. Blame tracking is a technique for propagating extra information through casts and proxies so that when a cast fails, the error message can point back to the original location of the cast in the source program.

The problem addressed by space-efficient casts also relates to higher-order casts. It turns out that in partially typed programs, a function or tuple can flow through very-many casts at runtime. With the approach described in this chapter, each cast adds another `lambda` wrapper or a tuple proxy. Not only does this take up considerable space, but it also makes the function calls and tuple operations slow. For example, a partially-typed version of quicksort could, in the worst case, build a chain of proxies of length $O(n)$ around the tuple, changing the overall time complexity of the algorithm from $O(n^2)$ to $O(n^3)$! Herman, Tomb, and Flanagan (2007) suggested a solution to this problem by representing casts using the coercion calculus of Henglein (1994), which prevents the creation of long chains of proxies by compressing them into a concise normal form. Siek, Thiemann, and Wadler (2015) give an algorithm for compressing coercions and Kuhlenschmidt, Almahallawi, and Siek (2019) show how to implement these ideas in the Grift compiler.

<https://github.com/Gradual-Typing/Grift>

There are also interesting interactions between gradual typing and other language features, such as parametric polymorphism, information-flow types, and type inference, to name a few. We recommend the reader to the online gradual typing bibliography:

<http://samth.github.io/gradual-typing-bib/>

**Figure 10.18**Diagram of the passes for $\mathcal{L}_?$ (gradual typing).

11

Parametric Polymorphism

This chapter studies the compilation of parametric polymorphism (aka. generics), compiling the $\mathcal{L}_{\text{Poly}}$ subset of Typed Racket. Parametric polymorphism enables programmers to make code more reusable by parameterizing functions and data structures with respect to the types that they operate on. For example, Figure 11.1 revisits the `map` example but this time gives it a more fitting type. This `map` function is parameterized with respect to the element type of the tuple. The type of `map` is the following polymorphic type as specified by the `All` and the type parameter `a`.

```
(All (a) ((a -> a) (Vector a a) -> (Vector a a)))
```

The idea is that `map` can be used at *all* choices of a type for parameter `a`. In Figure 11.1 we apply `map` to a tuple of integers, a choice of `Integer` for `a`, but we could have just as well applied `map` to a tuple of Booleans.

Figure 11.2 defines the concrete syntax of $\mathcal{L}_{\text{Poly}}$ and Figure 11.3 defines the abstract syntax. We add a second form for function definitions in which a type declaration comes before the `define`. In the abstract syntax, the return type in the `Def` is `Any`, but that should be ignored in favor of the return type in the type declaration. (The `Any` comes from using the same parser as in Chapter 9.) The presence of a type declaration enables the use of an `All` type for a function, thereby making it polymorphic. The grammar for types is extended to include polymorphic types and type variables.

By including polymorphic types in the *type* non-terminal we choose to make them first-class which has interesting repercussions on the compiler. Many languages with polymorphism, such as C++ (Stroustrup 1988) and Standard ML (Milner, Tofte,

```
(: map (All (a) ((a -> a) (Vector a a) -> (Vector a a))))  
(define (map f v)  
  (vector (f (vector-ref v 0)) (f (vector-ref v 1))))  
  
(define (inc [x : Integer]) : Integer (+ x 1))  
  
(vector-ref (map inc (vector 0 41)) 1)
```

Figure 11.1

The `map` example using parametric polymorphism.

<i>type</i>	::=	Integer
<i>exp</i>	::=	<i>int</i> (<i>read</i>) (<i>- exp</i>) (<i>+ exp exp</i>) (<i>- exp exp</i>)
<i>exp</i>	::=	<i>var</i> (<i>let</i> ([<i>var exp</i>]) <i>exp</i>)
<i>type</i>	::=	Boolean
<i>bool</i>	::=	#t #f
<i>cmp</i>	::=	eq? < <= > >=
<i>exp</i>	::=	<i>bool</i> (<i>and exp exp</i>) (<i>or exp exp</i>) (<i>not exp</i>) (<i>cmp exp exp</i>) (<i>if exp exp exp</i>)
<i>type</i>	::=	Void
<i>exp</i>	::=	(<i>set!</i> <i>var exp</i>) (<i>begin exp* exp</i>) (<i>while exp exp</i>) (<i>void</i>)
<i>type</i>	::=	(Vector <i>type*</i>)
<i>exp</i>	::=	(<i>vector exp*</i>) (<i>vector-length exp</i>) (<i>vector-ref exp int</i>) (<i>vector-set!</i> <i>exp int exp</i>)
<i>type</i>	::=	(<i>type ... -> type</i>)
<i>exp</i>	::=	(<i>exp exp ...</i>)
<i>def</i>	::=	(<i>define</i> (<i>var [var:type]</i> ...) : <i>type exp</i>)
<i>exp</i>	::=	(<i>lambda:</i> ([<i>var:type</i>] ...) : <i>type exp</i>) (<i>procedure-arity exp</i>)
<i>type</i>	::=	(<i>All</i> (<i>var ...</i>) <i>type</i>) <i>var</i>
<i>def</i>	::=	(: <i>var type</i>) (<i>define</i> (<i>var var ...</i>) <i>exp</i>)
$\mathcal{L}_{\text{Poly}}$::=	<i>def ... exp</i>

Figure 11.2

The concrete syntax of $\mathcal{L}_{\text{Poly}}$, extending \mathcal{L}_{λ} (Figure 8.3).

<i>type</i>	::=	Integer
<i>op</i>	::=	read + -
<i>exp</i>	::=	(Int <i>int</i>) (Prim <i>op</i> (<i>exp ...</i>))
<i>exp</i>	::=	(Var <i>var</i>) (Let <i>var exp exp</i>)
<i>type</i>	::=	Boolean
<i>bool</i>	::=	#t #f
<i>cmp</i>	::=	eq? < <= > >=
<i>op</i>	::=	<i>cmp</i> and or not
<i>exp</i>	::=	(Bool <i>bool</i>) (If <i>exp exp exp</i>)
<i>type</i>	::=	Void
<i>exp</i>	::=	(SetBang <i>var exp</i>) (Begin <i>exp* exp</i>) (WhileLoop <i>exp exp</i>) (Void)
<i>type</i>	::=	(Vector <i>type*</i>)
<i>op</i>	::=	vector vector-length
<i>exp</i>	::=	(Prim vector-ref (<i>exp</i> (Int <i>int</i>))) (Prim vector-set! (<i>exp</i> (Int <i>int</i>) <i>exp</i>)) (HasType <i>exp type</i>)
<i>type</i>	::=	(<i>type ... -> type</i>)
<i>exp</i>	::=	(Apply <i>exp exp ...</i>)
<i>def</i>	::=	(Def <i>var</i> ([<i>var:type</i>] ...) <i>type</i> '() <i>exp</i>)
<i>exp</i>	::=	(Lambda ([<i>var:type</i>] ...) <i>type exp</i>)
<i>op</i>	::=	procedure-arity
<i>type</i>	::=	(<i>All</i> (<i>var ...</i>) <i>type</i>) <i>var</i>
<i>def</i>	::=	(Decl <i>var type</i>) (Def <i>var</i> (<i>var ...</i>) 'Any '() <i>exp</i>)
$\mathcal{L}_{\text{Poly}}$::=	(ProgramDefsExp '() (<i>def ...</i>) <i>exp</i>)

Figure 11.3

The abstract syntax of $\mathcal{L}_{\text{Poly}}$, extending \mathcal{L}_{λ} (Figure 8.4).

```

(: apply-twice ((All (b) (b -> b)) -> Integer))
(define (apply-twice f)
  (if (f #t) (f 42) (f 777)))

(: id (All (a) (a -> a)))
(define (id x) x)

(apply-twice id)

```

Figure 11.4

An example illustrating first-class polymorphism.

and Harper 1990), only support second-class polymorphism, so it may be helpful to see an example of first-class polymorphism in action. In Figure 11.4 we define a function `apply-twice` whose parameter is a polymorphic function. The occurrence of a polymorphic type underneath a function type is enabled by the normal recursive structure of the grammar for *type* and the categorization of the `All` type as a *type*. The body of `apply-twice` applies the polymorphic function to a Boolean and to an integer.

The type checker for $\mathcal{L}_{\text{Poly}}$ in Figure 11.7 has three new responsibilities (compared to \mathcal{L}_{λ}). The type checking of function application is extended to handle the case where the operator expression is a polymorphic function. In that case the type arguments are deduced by matching the type of the parameters with the types of the arguments. The `match-types` auxiliary function carries out this deduction by recursively descending through a parameter type `pt` and the corresponding argument type `at`, making sure that they are equal except when there is a type parameter on the left (in the parameter type). If it is the first time that the type parameter has been encountered, then the algorithm deduces an association of the type parameter to the corresponding type on the right (in the argument type). If it is not the first time that the type parameter has been encountered, the algorithm looks up its deduced type and makes sure that it is equal to the type on the right. Once the type arguments are deduced, the operator expression is wrapped in an `Inst` AST node (for instantiate) that records the type of the operator, but more importantly, records the deduced type arguments. The return type of the application is the return type of the polymorphic function, but with the type parameters replaced by the deduced type arguments, using the `subst-type` function.

The second responsibility of the type checker to extend the `type-equal?` function to handle the `All` type. This is not quite as simple as for other types, such as function and tuple types, because two polymorphic types can be syntactically different even though they are equivalent types. For example, `(All (a) (a -> a))` is equivalent to `(All (b) (b -> b))`. Two polymorphic types should be considered equal if they differ only in the choice of the names of the type parameters. The `type-equal?` function in Figure 11.8 renames the type parameters of the first type to match the type parameters of the second type.

<i>type</i>	::=	... (All (var ...) type) var
<i>exp</i>	::=	... (Inst exp type (type ...))
<i>def</i>	::=	(Def var ([var:type] ...) type '() exp)
		(Poly (var ...) (Def var ([var:type] ...) type '() exp))
$\mathcal{L}_{\text{Inst}}$::=	(ProgramDfsExp '() (def ...) exp)

Figure 11.5

The abstract syntax of $\mathcal{L}_{\text{Inst}}$, extending \mathcal{L}_λ (Figure 8.4).

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

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

(vector-ref ((inst map (All (a) ((a -> a) (Vector a a) -> (Vector a a)))
  (Integer)))
  inc (vector 0 41)) 1)
```

Figure 11.6

Output of the type checker on the `map` example.

The third responsibility of the type checker is to make sure that only defined type variables appear in type annotations. The `check-well-formed` function defined in Figure 11.9 recursively inspects a type, making sure that each type variable has been defined.

The output language of the type checker is $\mathcal{L}_{\text{Inst}}$, defined in Figure 11.5. The type checker combines the type declaration and polymorphic function into a single definition, using the `Poly` form, to make polymorphic functions more convenient to process in next pass of the compiler.

The output of the type checker on the polymorphic `map` example is listed in Figure 11.6.


```

(define type-check-poly-class
  (class type-check-Lambda-class
    (super-new)
    (inherit check-type-equal?)

    (define/override (type-check-apply env e1 es)
      (define-values (e^ ty) ((type-check-exp env) e1))
      (define-values (es^ ty*) (for/lists (es^ ty*) ([e (in-list es)])
                                           ((type-check-exp env) e)))
      (match ty
        [(,ty^* ... -> ,rt)
         (for ([arg-ty ty*] [param-ty ty^*])
           (check-type-equal? arg-ty param-ty (Apply e1 es)))
         (values e^ es^ rt)]
        [(All ,xs (,tys ... -> ,rt))
         (define env^ (append (for/list ([x xs]) (cons x 'Type)) env))
         (define env^^ (for/fold ([env^^ env^]) ([arg-ty ty*] [param-ty tys])
                                   (match-types env^^ param-ty arg-ty)))
         (define targs
           (for/list ([x xs])
             (match (dict-ref env^^ x (lambda () #f))
               [#f (error 'type-check "type variable ~a not deduced\nin ~v"
                          x (Apply e1 es))]
               [ty ty])))
         (values (Inst e^ ty targs) es^ (subst-type env^^ rt))]
        [else (error 'type-check "expected a function, not ~a" ty)]))

    (define/override ((type-check-exp env) e)
      (match e
        [(Lambda `([,xs : ,Ts] ...) rT body)
         (for ([T Ts]) ((check-well-formed env) T))
         ((check-well-formed env) rT)
         ((super type-check-exp env) e)]
        [(HasType e1 ty)
         ((check-well-formed env) ty)
         ((super type-check-exp env) e)]
        [else ((super type-check-exp env) e)]))

    (define/override ((type-check-def env) d)
      (verbose 'type-check "poly/def" d)
      (match d
        [(Generic ts (Def f (and p:t* (list `[,xs : ,ps] ...)) rt info body))
         (define ts-env (for/list ([t ts]) (cons t 'Type)))
         (for ([p ps]) ((check-well-formed ts-env) p))
         ((check-well-formed ts-env) rt)
         (define new-env (append ts-env (map cons xs ps) env))
         (define-values (body^ ty^) ((type-check-exp new-env) body))
         (check-type-equal? ty^ rt body)
         (Generic ts (Def f p:t* rt info body^))]
        [else ((super type-check-def env) d)]))

    (define/override (type-check-program p)
      (match p
        [(Program info body)
         (type-check-program (ProgramDefsExp info '() body))]
        [(ProgramDefsExp info ds body)
         (define ds^ (combine-decls-defs ds))
         (define new-env (for/list ([d ds^])
                                   (cons (def-name d) (fun-def-type d))))
         (define ds^^ (for/list ([d ds^]) ((type-check-def new-env) d)))
         (define-values (body^ ty) ((type-check-exp new-env) body))
         (check-type-equal? ty 'Integer body)
         (ProgramDefsExp info ds^^ body^)]))
  ))

```

Figure 11.7

Type checker for the $\mathcal{L}_{\text{Poly}}$ language.

```

(define/override (type-equal? t1 t2)
  (match* (t1 t2)
    [(` (All ,xs ,T1) `(All ,ys ,T2))
     (define env (map cons xs ys))
     (type-equal? (subst-type env T1) T2)]
    [(other wise)
     (super type-equal? t1 t2)]))

(define/public (match-types env pt at)
  (match* (pt at)
    [(` (Integer 'Integer) env) [(` (Boolean 'Boolean) env)]
    [(` (Void 'Void) env) [(` (Any 'Any) env)]
    [(` (Vector ,pts ...) `(Vector ,ats ...))
     (for/fold ([env~ env]) ([pt1 pts] [at1 ats])
       (match-types env~ pt1 at1))]
    [(` (,pts ... -> ,prt) `(,ats ... -> ,art))
     (define env~ (match-types env prt art))
     (for/fold ([env^^ env~]) ([pt1 pts] [at1 ats])
       (match-types env^^ pt1 at1))]
    [(` (All ,pxs ,pt1) `(All ,axs ,at1))
     (define env~ (append (map cons pxs axs) env))
     (match-types env~ pt1 at1)]
    [(? symbol? x) at]
    (match (dict-ref env x (lambda () #f))
      [#f (error 'type-check "undefined type variable -a" x)]
      [Type (cons (cons x at) env)]
      [t~ (check-type-equal? at t~ 'matching env)])
    [(other wise) (error 'type-check "mismatch -a != a" pt at)]))

(define/public (subst-type env pt)
  (match pt
    [(Integer 'Integer) [(Boolean 'Boolean)]
    [(Void 'Void) [(Any 'Any)]
    [(Vector ,ts ...)
     `(Vector ,(for/list ([t ts]) (subst-type env t)))]
    [(,ts ... -> ,rt)
     `(,@(for/list ([t ts]) (subst-type env t)) -> ,(subst-type env rt))]
    [(All ,xs ,t)
     `(All ,xs ,(subst-type (append (map cons xs xs) env) t))]
    [(? symbol? x) (dict-ref env x)]
    [else (error 'type-check "expected a type not -a" pt)]))

(define/public (combine-decls-defs ds)
  (match ds
    [(` () '())
     [(` (, (Decl name type) . (, (Def f params _ info body) . ,ds~))
      (unless (equal? name f)
        (error 'type-check "name mismatch, -a != -a" name f))
      (match type
        [(` (All ,xs (,ps ... -> ,rt))
         (define params~ (for/list ([x params] [T ps]) `[x : ,T]))
         (cons (Generic xs (Def name params~ rt info body))
              (combine-decls-defs ds~))]
        [(,ps ... -> ,rt)
         (define params~ (for/list ([x params] [T ps]) `[x : ,T]))
         (cons (Def name params~ rt info body) (combine-decls-defs ds~))]
        [else (error 'type-check "expected a function type, not -a" type) ]])
     [(` (, (Def f params rt info body) . ,ds~)
      (cons (Def f params rt info body) (combine-decls-defs ds~)))]))

```

Figure 11.8

Auxiliary functions for type checking $\mathcal{L}_{\text{Poly}}$.

```

(match ty
  ['Integer (void)]
  ['Boolean (void)]
  ['Void (void)]
  [(? symbol? a)
   (match (dict-ref env a (lambda () #f))
     ['Type (void)]
     [else (error 'type-check "undefined type variable ~a" a)])]
  ['(Vector ,ts ...)
   (for ([t ts]) ((check-well-formed env) t)))]
  ['(,ts ... -> ,t)
   (for ([t ts]) ((check-well-formed env) t))
   ((check-well-formed env) t)]
  ['(All ,xs ,t)
   (define env~ (append (for/list ([x xs]) (cons x 'Type)) env))
   ((check-well-formed env~) t)]
  [else (error 'type-check "unrecognized type ~a" ty)]))

```

Figure 11.9

Well-formed types.

11.1 Compiling Polymorphism

Broadly speaking, there are four approaches to compiling parametric polymorphism, which we describe below.

Monomorphization generates a different version of a polymorphic function for each set of type arguments that it is used with, producing type-specialized code. This approach results in the most efficient code but requires whole-program compilation (no separate compilation) and increases code size. For our current purposes monomorphization is a non-starter because, with first-class polymorphism, it is sometimes not possible to determine which generic functions are used with which type arguments during compilation. (It can be done at runtime, with just-in-time compilation.) Monomorphization is used to compile C++ templates (Stroustrup 1988) and polymorphic functions in NESL (Blelloch et al. 1993) and ML (Weeks 2006).

Uniform representation generates one version of each polymorphic function but requires all values to have a common “boxed” format, such as the tagged values of type `Any` in \mathcal{L}_{Any} . Both polymorphic and non-polymorphic (i.e. monomorphic) code is compiled similarly to code in a dynamically typed language (like \mathcal{L}_{Dyn}), in which primitive operators require their arguments to be projected from `Any` and their results are injected into `Any`. (In object-oriented languages, the projection is accomplished via virtual method dispatch.) The uniform representation approach is compatible with separate compilation and with first-class polymorphism. However, it produces the least-efficient code because it introduces overhead in the entire program. This approach is used in implementations of CLU Liskov et al. 1979; Liskov 1993, ML (Cardelli 1984; Appel and MacQueen 1987), and Java (Bracha et al. 1998).

Mixed representation generates one version of each polymorphic function, using a boxed representation for type variables. Monomorphic code is compiled as usual (as in \mathcal{L}_{λ}) and conversions are performed at the boundaries between monomorphic and polymorphic (e.g. when a polymorphic function is instantiated and called). This approach is compatible with separate compilation and first-class polymorphism and maintains efficiency in monomorphic code. The trade off is increased overhead at the boundary between monomorphic and polymorphic code. This approach is used in implementations of ML (Leroy 1992) and Java, starting in Java 5 with the addition of autoboxing.

Type passing uses the unboxed representation in both monomorphic and polymorphic code. Each polymorphic function is compiled to a single function with extra parameters that describe the type arguments. The type information is used by the generated code to know how to access the unboxed values at runtime. This approach is used in implementation of the Napier88 language (Morrison et al. 1991) and ML (Harper and Morrisett 1995). Type passing is compatible with separate compilation and first-class polymorphism and maintains the efficiency for monomorphic code. There is runtime overhead in polymorphic code from dispatching on type information.

```

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

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

(vector-ref ((cast map
  ((Any -> Any) (Vector Any Any) -> (Vector Any Any))
  ((Integer -> Integer) (Vector Integer Integer)
    -> (Vector Integer Integer)))
  inc (vector 0 41)) 1)

```

Figure 11.10

The polymorphic `map` example after type erasure.

In this chapter we use the mixed representation approach, partly because of its favorable attributes, and partly because it is straightforward to implement using the tools that we have already built to support gradual typing. To compile polymorphic functions, we add just one new pass, `erase-types`, to compile $\mathcal{L}_{\text{Inst}}$ to $\mathcal{L}_{\text{Cast}}$.

11.2 Erase Types

We use the `Any` type from Chapter 9 to represent type variables. For example, Figure 11.10 shows the output of the `erase-types` pass on the polymorphic `map` (Figure 11.1). The occurrences of type parameter `a` are replaced by `Any` and the polymorphic `All` types are removed from the type of `map`.

This process of type erasure creates a challenge at points of instantiation. For example, consider the instantiation of `map` in Figure 11.6. The type of `map` is

```
(All (a) ((a -> a) (Vector a a) -> (Vector a a)))
```

and it is instantiated to

```
((Integer -> Integer) (Vector Integer Integer)
 -> (Vector Integer Integer))
```

After erasure, the type of `map` is

```
((Any -> Any) (Vector Any Any) -> (Vector Any Any))
```

but we need to convert it to the instantiated type. This is easy to do in the language $\mathcal{L}_{\text{Cast}}$ with a single `cast`. In Figure 11.10, the instantiation of `map` has been compiled to a `cast` from the type of `map` to the instantiated type. The source and target type of a cast must be consistent (Figure 10.4), which indeed is the case because both the source and target are obtained from the same polymorphic type of `map`, replacing the type parameters with `Any` in the former and with the deduced type arguments in the later. (Recall that the `Any` type is consistent with any type.)

To implement the `erase-types` pass, we recommend defining a recursive auxiliary function named `erase-type` that applies the following two transformations. It replaces type variables with `Any`

x
 \Rightarrow
Any

and it removes the polymorphic **All** types.

$(\mathbf{All} \ xs \ T_1)$
 \Rightarrow
 T'_1

Apply the **erase-type** function to all of the type annotations in the program.

Regarding the translation of expressions, the case for **Inst** is the interesting one. We translate it into a **Cast**, as shown below. The type of the subexpression e is the polymorphic type $(\mathbf{All} \ xs \ T)$. The source type of the cast is the erasure of T , the type T' . The target type T'' is the result of substituting the argument types ts for the type parameters xs in T followed by doing type erasure.

$(\mathbf{Inst} \ e \ (\mathbf{All} \ xs \ T) \ ts)$
 \Rightarrow
 $(\mathbf{Cast} \ e' \ T' \ T'')$

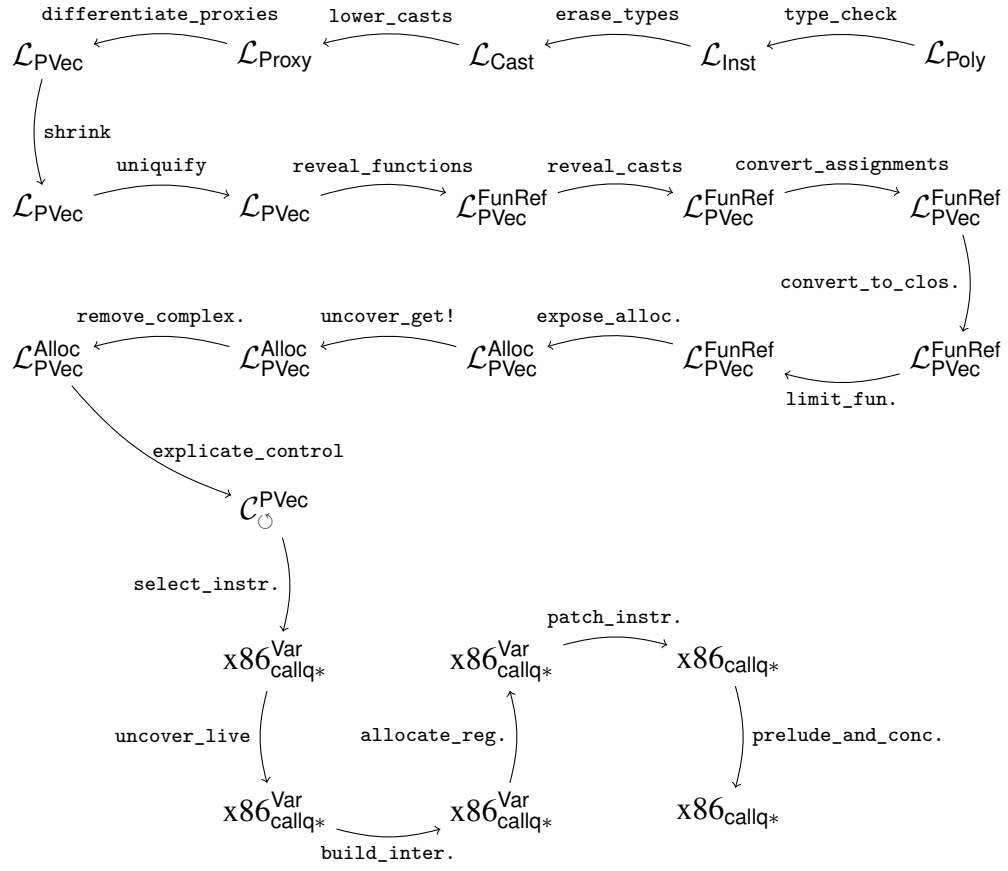
where $T'' = (\mathbf{erase-type} \ (\mathbf{subst-type} \ s \ T))$ and $s = (\mathbf{map} \ \mathbf{cons} \ xs \ ts)$.

Finally, each polymorphic function is translated to a regular function in which type erasure has been applied to all the type annotations and the body.

$(\mathbf{Poly} \ ts \ (\mathbf{Def} \ f \ ([x_1 : T_1] \ \dots) \ T_r \ \mathbf{info} \ e))$
 \Rightarrow
 $(\mathbf{Def} \ f \ ([x_1 : T'_1] \ \dots) \ T'_r \ \mathbf{info} \ e')$

Exercise 39 Implement a compiler for the polymorphic language $\mathcal{L}_{\text{Poly}}$ by extending and adapting your compiler for $\mathcal{L}_?$. Create 6 new test programs that use polymorphic functions. Some of them should make use of first-class polymorphism.

Figure 11.11 provides an overview of the passes needed to compile $\mathcal{L}_{\text{Poly}}$.

**Figure 11.11**Diagram of the passes for $\mathcal{L}_{\text{Poly}}$ (parametric polymorphism).

A Appendix

A.1 Interpreters

We provide interpreters for each of the source languages \mathcal{L}_{int} , \mathcal{L}_{var} , ... in the files `interp_Lint.rkt`, `interp-Lvar.rkt`, etc. The interpreters for the intermediate languages \mathcal{C}_{var} and \mathcal{C}_{ff} are in `interp-Cvar.rkt` and `interp-C1.rkt`. The interpreters for \mathcal{C}_{tup} , \mathcal{C}_{fun} , pseudo-x86, and x86 are in the `interp.rkt` file.

A.2 Utility Functions

The utility functions described in this section are in the `utilities.rkt` file of the support code.

interp-tests The `interp-tests` function runs the compiler passes and the interpreters on each of the specified tests to check whether each pass is correct. The `interp-tests` function has the following parameters:

name (a string) a name to identify the compiler,

typechecker a function of exactly one argument that either raises an error using the `error` function when it encounters a type error, or returns `#f` when it encounters a type error. If there is no type error, the type checker returns the program.

passes a list with one entry per pass. An entry is a list with four things:

1. a string giving the name of the pass,
2. the function that implements the pass (a translator from AST to AST),
3. a function that implements the interpreter (a function from AST to result value) for the output language,
4. and a type checker for the output language. Type checkers for the *R* and *C* languages are provided in the support code. For example, the type checkers for \mathcal{L}_{var} and \mathcal{C}_{var} are in `type-check-Lvar.rkt` and `type-check-Cvar.rkt`. The type checker entry is optional. The support code does not provide type checkers for the x86 languages.

source-interp an interpreter for the source language. The interpreters from Appendix A.1 make a good choice.

test-family (a string) for example, "r1", "r2", etc.

tests a list of test numbers that specifies which tests to run. (see below)

The `interp-tests` function assumes that the subdirectory `tests` has a collection of Racket programs whose names all start with the family name, followed by an underscore and then the test number, ending with the file extension `.rkt`. Also, for each test program that calls `read` one or more times, there is a file with the same name except that the file extension is `.in` that provides the input for the Racket program. If the test program is expected to fail type checking, then there should be an empty file of the same name but with extension `.tyerr`.

compiler-tests runs the compiler passes to generate x86 (a `.s` file) and then runs the GNU C compiler (gcc) to generate machine code. It runs the machine code and checks that the output is 42. The parameters to the `compiler-tests` function are similar to those of the `interp-tests` function, and consist of

- a compiler name (a string),
- a type checker,
- description of the passes,
- name of a test-family, and
- a list of test numbers.

compile-file takes a description of the compiler passes (see the comment for **interp-tests**) and returns a function that, given a program file name (a string ending in **.rkt**), applies all of the passes and writes the output to a file whose name is the same as the program file name but with **.rkt** replaced with **.s**.

read-program takes a file path and parses that file (it must be a Racket program) into an abstract syntax tree.

parse-program takes an S-expression representation of an abstract syntax tree and converts it into the struct-based representation.

assert takes two parameters, a string (**msg**) and Boolean (**bool**), and displays the message **msg** if the Boolean **bool** is false.

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

A.3 x86 Instruction Set Quick-Reference

Table A.1 lists some x86 instructions and what they do. We write $A \rightarrow B$ 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
<code>addq A, B</code>	$A + B \rightarrow B$
<code>negq A</code>	$-A \rightarrow A$
<code>subq A, B</code>	$B - A \rightarrow B$
<code>imulq A, B</code>	$A \times B \rightarrow B$
<code>callq L</code>	Pushes the return address and jumps to label L
<code>callq *A</code>	Calls the function at the address A .
<code>retq</code>	Pops the return address and jumps to it
<code>popq A</code>	$*rsp \rightarrow A; rsp + 8 \rightarrow rsp$
<code>pushq A</code>	$rsp - 8 \rightarrow rsp; A \rightarrow *rsp$
<code>leaq A, B</code>	$A \rightarrow B$ (B must be a register)
<code>cmpq A, B</code>	compare A and B and set the flag register (B must not be an immediate)
<code>je L</code>	Jump to label L if the flag register matches the condition code of the instruction, otherwise go to the next instructions. The condition codes are e for “equal”, l for “less”, le for “less or equal”, g for “greater”, and ge for “greater or equal”.
<code>jle L</code>	
<code>jg L</code>	
<code>jge L</code>	
<code>jmp L</code>	Jump to label L
<code>movq A, B</code>	$A \rightarrow B$
<code>movzbq A, B</code>	$A \rightarrow B$, where A is a single-byte register (e.g., <code>al</code> or <code>cl</code>), B is a 8-byte register, and the extra bytes of B are set to zero.
<code>notq A</code>	$\sim A \rightarrow A$ (bitwise complement)
<code>orq A, B</code>	$A B \rightarrow B$ (bitwise-or)
<code>andq A, B</code>	$A \& B \rightarrow B$ (bitwise-and)
<code>salq A, B</code>	$B \ll A \rightarrow B$ (arithmetic shift left, where A is a constant)
<code>sarq A, B</code>	$B \gg A \rightarrow B$ (arithmetic shift right, where A is a constant)
<code>sete A</code>	If the flag matches the condition code, then $1 \rightarrow A$, else $0 \rightarrow A$. Refer to <code>je</code> above for the description of the condition codes. A must be a single byte register (e.g., <code>al</code> or <code>cl</code>).
<code>setl A</code>	
<code>setg A</code>	
<code>setge A</code>	

Table A.1

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

References

- Abelson, Harold, and Gerald J. Sussman. 1996. *Structure and Interpretation of Computer Programs*. 2nd. Cambridge, MA, USA: MIT Press. ISBN: 0262011530.
- Aho, Alfred V., Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0321486811.
- Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: principles, techniques, and tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0-201-10088-6.
- Allen, Frances E. 1970. "Control flow analysis." In *Proceedings of a symposium on Compiler optimization*, 1–19.
- Anderson, Christopher, and Sophia Drossopoulou. 2003. "BabyJ - From Object Based to Class Based Programming via Types." In *WOOD '03*, vol. 82. 8. Elsevier.
- Appel, Andrew W. 1989. "Runtime tags aren't necessary" [in English]. *LISP and Symbolic Computation* 2 (2): 153–162. ISSN: 0892-4635. <https://doi.org/10.1007/BF01811537>. <http://dx.doi.org/10.1007/BF01811537>.
- Appel, Andrew W. 1990. "A runtime system." *LISP and Symbolic Computation* 3, no. 4 (November): 343–380.
- Appel, Andrew W. 1991. *Compiling with Continuations*. Cambridge University Press. <https://doi.org/10.1017/CBO9780511609619>.
- Appel, Andrew W., and David B. MacQueen. 1987. "A standard ML compiler." In *Functional Programming Languages and Computer Architecture: Portland, Oregon, USA, September 14–16, 1987 Proceedings*, edited by Gilles Kahn, 301–324. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN: 978-3-540-47879-9. https://doi.org/10.1007/3-540-18317-5_17. http://dx.doi.org/10.1007/3-540-18317-5_17.
- Appel, Andrew W., and Jens Palsberg. 2003. *Modern Compiler Implementation in Java*. Cambridge University Press. ISBN: 052182060X.
- Backus, J. W., F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, et al. 1960. "Report on the Algorithmic Language ALGOL 60." Edited by Peter Naur. *Commun. ACM* (New York, NY, USA) 3, no. 5 (May): 299–314. ISSN: 0001-0782. <https://doi.org/10.1145/367236.367262>. <http://doi.acm.org/10.1145/367236.367262>.
- Backus, John. 1978. "The History of Fortran I, II, and III." In *History of Programming Languages*, 25–74. New York, NY, USA: Association for Computing Machinery. ISBN: 0127450408. <https://doi.org/10.1145/800025.1198345>.
- Baker, J., A. Cunei, T. Kalibera, F. Pizlo, and J. Vitek. 2009. "Accurate Garbage Collection in Uncooperative Environments Revisited." *Concurr. Comput. : Pract. Exper.* (Chichester, UK) 21, no. 12 (August): 1572–1606. ISSN: 1532-0626. <https://doi.org/10.1002/cpe.v21:12>. <http://dx.doi.org/10.1002/cpe.v21:12>.
- Balakrishnan, V. K. 1996. *Introductory Discrete Mathematics*. Dover Publications, Incorporated. ISBN: 0486691152.
- Barry, Paul. 2016. *Head First Python*. O'Reilly.
- Blackburn, Stephen M., Perry Cheng, and Kathryn S. McKinley. 2004. "Myths and Realities: The Performance Impact of Garbage Collection." In *Proceedings of the Joint International Conference*

- on *Measurement and Modeling of Computer Systems*, 25–36. SIGMETRICS '04/Performance '04. New York, NY, USA: ACM. ISBN: 1-58113-873-3. <https://doi.org/10.1145/1005686.1005693>. <http://doi.acm.org/10.1145/1005686.1005693>.
- Blelloch, Guy E., Jonathan C. Hardwick, Siddhartha Chatterjee, Jay Sipelstein, and Marco Zagha. 1993. "Implementation of a Portable Nested Data-Parallel Language." In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 102–111. PPOPP '93. New York, NY, USA: Association for Computing Machinery. ISBN: 0897915895. <https://doi.org/10.1145/155332.155343>. <https://doi.org/10.1145/155332.155343>.
- Bracha, Gilad, Martin Odersky, David Stoutamire, and Philip Wadler. 1998. "Making the future safe for the past: adding genericity to the Java programming language." In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 183–200. OOPSLA '98. New York, NY, USA: ACM. ISBN: 1-58113-005-8. <https://doi.org/http://doi.acm.org/10.1145/286936.286957>. <http://doi.acm.org/10.1145/286936.286957>.
- Brélaz, Daniel. 1979. "New methods to color the vertices of a graph." *Commun. ACM* (New York, NY, USA) 22 (4): 251–256. ISSN: 0001-0782.
- Briggs, Preston, Keith D. Cooper, and Linda Torczon. 1994. "Improvements to graph coloring register allocation." *ACM Trans. Program. Lang. Syst.* 16 (3): 428–455. ISSN: 0164-0925.
- Bryant, Randal E., and David R. O'Hallaron. 2005. *x86-64 Machine-Level Programming*. Carnegie Mellon University, September.
- Bryant, Randal E., and David R. O'Hallaron. 2010. *Computer Systems: A Programmer's Perspective*. 2nd. USA: Addison-Wesley Publishing Company. ISBN: 0136108040.
- Cardelli, Luca. 1983. *The Functional Abstract Machine*. Technical report TR-107. AT&T Bell Laboratories.
- Cardelli, Luca. 1984. "Compiling a Functional Language." In *ACM Symposium on LISP and Functional Programming*, 208–217. LFP '84. ACM.
- Cardelli, Luca, and Peter Wegner. 1985. "On understanding types, data abstraction, and polymorphism." *ACM Comput. Surv.* (New York, NY, USA) 17 (4): 471–523. ISSN: 0360-0300.
- Chaitin, G. J. 1982. "Register allocation & spilling via graph coloring." In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, 98–105. ACM Press. ISBN: 0-89791-074-5.
- Chaitin, Gregory J., Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. 1981. "Register allocation via coloring." *Computer Languages* 6:47–57.
- Cheney, C. J. 1970. "A Nonrecursive List Compacting Algorithm." *Communications of the ACM* 13 (11).
- Chow, Frederick, and John Hennessy. 1984. "Register allocation by priority-based coloring." In *SIGPLAN '84: Proceedings of the 1984 SIGPLAN symposium on Compiler construction*, 222–232. ACM Press. ISBN: 0-89791-139-3.
- Church, Alonzo. 1932. "A Set of Postulates for the Foundation of Logic" [in English]. *Annals of Mathematics*, Second Series, 33 (2): pp. 346–366. ISSN: 0003486X. <http://www.jstor.org/stable/1968337>.
- Clarke, Keith. 1989. "One-Pass Code Generation Using Continuations." *Softw. Pract. Exper.* (USA) 19, no. 12 (November): 1175–1192. ISSN: 0038-0644.
- Collins, George E. 1960. "A Method for Overlapping and Erasure of Lists." *Commun. ACM* (New York, NY, USA) 3, no. 12 (December): 655–657. ISSN: 0001-0782. <https://doi.org/10.1145/367487.367501>. <https://doi.org/10.1145/367487.367501>.
- Cooper, Keith, and Linda Torczon. 2011. *Engineering a Compiler*. 2nd. Morgan Kaufmann.
- Cooper, Keith D., and L. Taylor Simpson. 1998. "Live range splitting in a graph coloring register allocator." In *Compiler Construction*.
- Cormen, Thomas H., Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. 2001. *Introduction to Algorithms*. McGraw-Hill Higher Education. ISBN: 0070131511.
- CPython github repository*. 2021. <https://github.com/python/cpython>. Python Software Foundation.

- Cutler, Cody, and Robert Morris. 2015. "Reducing Pause Times with Clustered Collection." In *Proceedings of the 2015 International Symposium on Memory Management*, 131–142. ISMM '15. New York, NY, USA: ACM. ISBN: 978-1-4503-3589-8. <https://doi.org/10.1145/2754169.2754184>. <http://doi.acm.org/10.1145/2754169.2754184>.
- Danvy, Olivier. 1991. *Three Steps for the CPS Transformation*. Technical report CIS-92-02. Kansas State University, December.
- Danvy, Olivier. 2003. "A New One-Pass Transformation into Monadic Normal Form." In *Compiler Construction*, 2622:77–89. LNCS.
- Detlefs, David, Christine Flood, Steve Heller, and Tony Printezis. 2004. "Garbage-first Garbage Collection." In *Proceedings of the 4th International Symposium on Memory Management*, 37–48. ISMM '04. New York, NY, USA: ACM. ISBN: 1-58113-945-4. <https://doi.org/10.1145/1029873.1029879>. <http://doi.acm.org/10.1145/1029873.1029879>.
- Dieckmann, Sylvia, and Urs Hölzle. 1999. "A Study of the Allocation Behavior of the SPECjvm98 Java Benchmark." In *ECOOP'99 - Object-Oriented Programming, 13th European Conference, Lisbon, Portugal, June 14-18, 1999, Proceedings*, edited by Rachid Guerraoui, 1628:92–115. Lecture Notes in Computer Science. Springer. https://doi.org/10.1007/3-540-48743-3_5. https://doi.org/10.1007/3-540-48743-3_5.
- Dijkstra, E. W. 1982. *Why numbering should start at zero*. Technical report EWD831. University of Texas at Austin.
- Diwan, Amer, Eliot Moss, and Richard Hudson. 1992. "Compiler Support for Garbage Collection in a Statically Typed Language." In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, 273–282. PLDI '92. New York, NY, USA: ACM. ISBN: 0-89791-475-9. <https://doi.org/10.1145/143095.143140>. <http://doi.acm.org/10.1145/143095.143140>.
- Dunfield, Jana, and Neel Krishnaswami. 2021. "Bidirectional Typing." *ACM Comput. Surv.* (New York, NY, USA) 54, no. 5 (May). ISSN: 0360-0300. <https://doi.org/10.1145/3450952>. <https://doi.org/10.1145/3450952>.
- Dybvig, R. Kent. 1987a. *The SCHEME Programming Language*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc. ISBN: 0-13-791864-X.
- Dybvig, R. Kent. 1987b. "Three Implementation Models for Scheme." PhD diss., University of North Carolina at Chapel Hill.
- Dybvig, R. Kent. 2006. "The Development of Chez Scheme." In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, 1–12. ICFP '06. New York, NY, USA: ACM. ISBN: 1-59593-309-3. <https://doi.org/10.1145/1159803.1159805>. <http://doi.acm.org/10.1145/1159803.1159805>.
- Dybvig, R. Kent, and Andrew Keep. 2010. *P523 Compiler Assignments*. Technical report. Indiana University.
- Felleisen, Matthias, M.D. Barski Conrad, David Van Horn, and Eight Students of Northeastern University. 2013. *Realm of Racket: Learn to Program, One Game at a Time!* San Francisco, CA, USA: No Starch Press. ISBN: 1593274912.
- Felleisen, Matthias, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2001. *How to Design Programs: An Introduction to Programming and Computing*. Cambridge, MA, USA: MIT Press. ISBN: 0-262-06218-6.
- Fischer, Michael J. 1972. "Lambda Calculus Schemata." In *Proceedings of ACM Conference on Proving Assertions about Programs*, 104–109. Las Cruces, New Mexico, USA: Association for Computing Machinery. ISBN: 9781450378918. <https://doi.org/10.1145/800235.807077>. <https://doi.org/10.1145/800235.807077>.
- Flanagan, Cormac. 2006. "Hybrid Type Checking." In *POPL 2006: The 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 245–256. Charleston, South Carolina, January.
- Flanagan, Cormac, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. "The essence of compiling with continuations." In *Conference on Programming Language Design and Implementation*, 502–514. PLDI. June.

- Flatt, Matthew, Caner Derici, R. Kent Dybvig, Andrew W. Keep, Gustavo E. Massaccesi, Sarah Spall, Sam Tobin-Hochstadt, and Jon Zeppieri. 2019. "Rebuilding Racket on Chez Scheme (Experience Report)." *Proc. ACM Program. Lang.* (New York, NY, USA) 3, no. ICFP (July). <https://doi.org/10.1145/3341642>. <https://doi.org/10.1145/3341642>.
- Flatt, Matthew, Robert Bruce Findler, and PLT. 2014. *The Racket Guide*. Technical report 6.0. PLT Inc.
- Flatt, Matthew, and PLT. 2014. *The Racket Reference 6.0*. Technical report. [Http://docs.racket-lang.org/reference/index.html](http://docs.racket-lang.org/reference/index.html). PLT Inc.
- Friedman, Daniel P., and Matthias Felleisen. 1996. *The Little Schemer (4th Ed.)* Cambridge, MA, USA: MIT Press. ISBN: 0-262-56099-2.
- Friedman, Daniel P., Mitchell Wand, and Christopher T. Haynes. 2001. *Essentials of programming languages (2. ed.)* MIT Press. ISBN: 978-0-262-06217-6.
- Friedman, Daniel P., and David S. Wise. 1976. *Cons should not evaluate its arguments*. Technical report TR44. Indiana University.
- Gamari, Ben, and Laura Dietz. 2020. "Alligator Collector: A Latency-Optimized Garbage Collector for Functional Programming Languages." In *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management*, 87–99. ISMM 2020. New York, NY, USA: Association for Computing Machinery. ISBN: 9781450375665. <https://doi.org/10.1145/3381898.3397214>. <https://doi.org/10.1145/3381898.3397214>.
- Gebremedhin, Assefaw Hadish. 1999. "Parallel Graph Coloring." PhD diss., University of Bergen.
- George, Lal, and Andrew W. Appel. 1996. "Iterated Register Coalescing." *ACM Trans. Program. Lang. Syst.* (New York, NY, USA) 18, no. 3 (May): 300–324. ISSN: 0164-0925. <https://doi.org/10.1145/229542.229546>. <https://doi.org/10.1145/229542.229546>.
- Ghuloum, Abdulaziz. 2006. "An Incremental Approach to Compiler Construction." In *Scheme and Functional Programming Workshop*.
- Gilray, Thomas, Steven Lyde, Michael D. Adams, Matthew Might, and David Van Horn. 2016. "Pushdown Control-Flow Analysis for Free." In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 691–704. POPL '16. New York, NY, USA: Association for Computing Machinery. ISBN: 9781450335492. <https://doi.org/10.1145/2837614.2837631>. <https://doi.org/10.1145/2837614.2837631>.
- Goldberg, Benjamin. 1991. "Tag-free Garbage Collection for Strongly Typed Programming Languages." In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, 165–176. PLDI '91. New York, NY, USA: ACM. ISBN: 0-89791-428-7. <https://doi.org/10.1145/113445.113460>. <http://doi.acm.org/10.1145/113445.113460>.
- Gordon, M., R. Milner, L. Morris, M. Newey, and C. Wadsworth. 1978. "A Metalanguage for Interactive Proof in LCF." In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 119–130. POPL '78. New York, NY, USA: Association for Computing Machinery. ISBN: 9781450373487. <https://doi.org/10.1145/512760.512773>. <https://doi.org/10.1145/512760.512773>.
- Gronski, Jessica, Kenneth Knowles, Aaron Tomb, Stephen N. Freund, and Cormac Flanagan. 2006. "Sage: Hybrid Checking for Flexible Specifications." In *Scheme and Functional Programming Workshop*, 93–104.
- Harper, Robert. 2016. *Practical Foundations for Programming Languages*. 2nd. Cambridge University Press.
- Harper, Robert, and Greg Morrisett. 1995. "Compiling polymorphism using intensional type analysis." In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 130–141. ACM Press. ISBN: 0-89791-692-1.
- Hatcliff, John, and Olivier Danvy. 1994. "A generic account of continuation-passing styles." In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 458–471. Portland, Oregon, United States: ACM Press. ISBN: 0-89791-636-0.
- Henderson, Fergus. 2002. "Accurate Garbage Collection in an Uncooperative Environment." In *Proceedings of the 3rd International Symposium on Memory Management*, 150–156. ISMM '02. New York, NY, USA: ACM. ISBN: 1-58113-539-4. <https://doi.org/10.1145/512429.512449>. <http://doi.acm.org/10.1145/512429.512449>.

- Henglein, Fritz. 1994. “Dynamic typing: syntax and proof theory.” *Science of Computer Programming* 22, no. 3 (June): 197–230.
- Herman, David, Aaron Tomb, and Cormac Flanagan. 2007. “Space-Efficient Gradual Typing.” In *Trends in Functional Prog. (TFP)*, XXVIII. April.
- Herman, David, Aaron Tomb, and Cormac Flanagan. 2010. “Space-efficient gradual typing” [in English]. *Higher-Order and Symbolic Computation* 23 (2): 167–189.
- Horwitz, L. P., R. M. Karp, R. E. Miller, and S. Winograd. 1966. “Index Register Allocation.” *J. ACM* (New York, NY, USA) 13, no. 1 (January): 43–61. ISSN: 0004-5411. <https://doi.org/10.1145/321312.321317>.
- Intel. 2015. *Intel 64 and IA-32 Architectures Software Developer’s Manual Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B, 3C and 3D*. December.
- Jacek, Nicholas, and J. Eliot B. Moss. 2019. “Learning When to Garbage Collect with Random Forests.” In *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management*, 53–63. ISMM 2019. New York, NY, USA: Association for Computing Machinery. ISBN: 9781450367226. <https://doi.org/10.1145/3315573.3329983>.
- Jones, Neil D., Carsten K. Gomard, and Peter Sestoft. 1993. *Partial evaluation and automatic program generation*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc. ISBN: 0-13-020249-5.
- Jones, Richard, Antony Hosking, and Eliot Moss. 2011. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. 1st. Chapman & Hall/CRC. ISBN: 1420082795.
- Jones, Richard, and Rafael Lins. 1996. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. New York, NY, USA: John Wiley & Sons, Inc. ISBN: 0-471-94148-4.
- Keep, Andrew W. 2012. “A Nanopass Framework for Commercial Compiler Development.” PhD diss., Indiana University, December.
- Keep, Andrew W., Alex Hearn, and R. Kent Dybvig. 2012. “Optimizing Closures in O(0)-time.” In *Proceedings of the 2012 Workshop on Scheme and Functional Programming*. Scheme ’12.
- Kelsey, R., W. Clinger, and J. Rees (eds.) 1998. “Revised⁵ Report on the Algorithmic Language Scheme.” *Higher-Order and Symbolic Computation* 11, no. 1 (August).
- Kempe, A. B. 1879. “On the Geographical Problem of the Four Colours.” *American Journal of Mathematics* 2 (3): 193–200. ISSN: 00029327, 10806377. <http://www.jstor.org/stable/2369235>.
- Kernighan, Brian W., and Dennis M. Ritchie. 1988. *The C programming language*. Upper Saddle River, NJ, USA: Prentice Hall Press. ISBN: 0-13-110362-8.
- Kildall, Gary A. 1973. “A unified approach to global program optimization.” In *POPL ’73: Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 194–206. ACM Press.
- Kleene, S. 1952. *Introduction to Metamathematics*.
- Knuth, Donald E. 1964. “Backus Normal Form vs. Backus Naur Form.” *Commun. ACM* (New York, NY, USA) 7, no. 12 (December): 735–736. ISSN: 0001-0782. <https://doi.org/10.1145/355588.365140>.
- Kuhlenschmidt, Andre, Deyaaeldeen Almahallawi, and Jeremy G. Siek. 2019. “Toward Efficient Gradual Typing for Structural Types via Coercions.” In *Conference on Programming Language Design and Implementation*. PLDI. ACM, June.
- Lawall, Julia L., and Olivier Danvy. 1993. “Separating Stages in the Continuation-Passing Style Transformation.” In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 124–136. POPL ’93. Charleston, South Carolina, USA: Association for Computing Machinery. ISBN: 0897915607. <https://doi.org/10.1145/158511.158613>.
- Lehtosalo, Jukka. 2021. *MyPy Optional Type Checker for Python*. <http://mypy-lang.org/>, June.
- Leroy, Xavier. 1992. “Unboxed objects and polymorphic typing.” In *POPL ’92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 177–188. New York, NY, USA: ACM Press. ISBN: 0-89791-453-8.

- Lieberman, Henry, and Carl Hewitt. 1983. "A Real-time Garbage Collector Based on the Lifetimes of Objects." *Commun. ACM* (New York, NY, USA) 26, no. 6 (June): 419–429. ISSN: 0001-0782. <https://doi.org/10.1145/358141.358147>. <http://doi.acm.org/10.1145/358141.358147>.
- Liskov, Barbara. 1993. "A history of CLU." In *HOPL-II: The second ACM SIGPLAN conference on History of programming languages*, 133–147. New York, NY, USA: ACM. ISBN: 0-89791-570-4.
- Liskov, Barbara, Russ Atkinson, Toby Bloom, Eliot Moss, Craig Schaffert, Bob Scheifler, and Alan Snyder. 1979. *CLU Reference Manual*. Technical report LCS-TR-225. MIT, October.
- Logothetis, George, and Prateek Mishra. 1981. "Compiling short-circuit boolean expressions in one pass." *Software: Practice and Experience* 11 (11): 1197–1214. <https://doi.org/https://doi.org/10.1002/spe.4380111104>. eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380111104>. <https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380111104>.
- Lutz, Mark. 2013. *Learning Python*. 5th. O'Reilly.
- Matthes, Eric. 2019. *Python Crash Course*. 2nd. No Starch Press.
- Matthews, Jacob, and Robert Bruce Findler. 2007. "Operational Semantics for Multi-Language Programs." In *The 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. January.
- Matula, David W., George Marble, and Joel D. Isaacson. 1972. "GRAPH COLORING ALGORITHM." This research was supported in part by the Advanced Research Projects Agency of the Department of Defense under contract SD-302 and by the National Science Foundation under contract GJ-446." In *Graph Theory and Computing*, edited by RONALD C. READ, 109–122. Academic Press. ISBN: 978-1-4832-3187-7. <https://doi.org/https://doi.org/10.1016/B978-1-4832-3187-7.50015-5>. <http://www.sciencedirect.com/science/article/pii/S009753977500155>.
- Matz, Michael, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. 2013. *System V Application Binary Interface, AMD64 Architecture Processor Supplement*. October.
- McCarthy, John. 1960. "Recursive functions of symbolic expressions and their computation by machine, Part I." *Commun. ACM* (New York, NY, USA) 3 (4): 184–195. ISSN: 0001-0782.
- Microsoft. 2018. *x64 Architecture*. <https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/x64-architecture>, March.
- Microsoft. 2020. *x64 calling convention*. <https://docs.microsoft.com/en-us/cpp/build/x64-calling-convention>, July.
- Milner, Robin, Mads Tofte, and Robert Harper. 1990. *The definition of Standard ML*. MIT Press. ISBN: 0-262-63132-6.
- Minamide, Yasuhiko, Greg Morrisett, and Robert Harper. 1996. "Typed closure conversion." In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 271–283. POPL '96. New York, NY, USA: ACM. ISBN: 0-89791-769-3. <https://doi.org/http://doi.acm.org/10.1145/237721.237791>. <http://doi.acm.org/10.1145/237721.237791>.
- Moggi, Eugenio. 1991. "Notions of computation and monads." *Inf. Comput.* (Duluth, MN, USA) 93 (1): 55–92. ISSN: 0890-5401.
- Moore, E.F. 1959. "The shortest path through a maze." In *Proceedings of an International Symposium on the Theory of Switching*. April.
- Morrison, R., A. Dearle, R. C. H. Connor, and A. L. Brown. 1991. "An Ad Hoc Approach to the Implementation of Polymorphism." *ACM Trans. Program. Lang. Syst.* (New York, NY, USA) 13, no. 3 (July): 342–371. ISSN: 0164-0925. <https://doi.org/10.1145/117009.117017>. <http://doi.acm.org/10.1145/117009.117017>.
- Al-Omari, Hussein, and Khair Eddin Sabri. 2006. "New Graph Coloring Algorithms." *Journal of Mathematics and Statistics* 2 (4).
- Österlund, Erik, and Welf Löwe. 2016. "Block-Free Concurrent GC: Stack Scanning and Copying." In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, 1–12. ISMM 2016. New York, NY, USA: Association for Computing Machinery. ISBN: 9781450343176. <https://doi.org/10.1145/2926697.2926701>. <https://doi.org/10.1145/2926697.2926701>.
- Palsberg, Jens. 2007. "Register allocation via coloring of chordal graphs." In *CATS '07: Proceedings of the thirteenth Australasian symposium on Theory of computing*, 3–3. Darlinghurst, Australia, Australia: Australian Computer Society, Inc. ISBN: 1-920-68246-5.

- Peyton Jones, Simon L., and André L. M. Santos. 1998. “A transformation-based optimiser for Haskell.” *Science of Computer Programming* 32 (1): 3–47.
- Pierce, Benjamin C. 2002. *Types and Programming Languages*. MIT Press.
- Pierce, Benjamin C., ed. 2004. *Advanced Topics in Types and Programming Languages*. The MIT press.
- Pierce, Benjamin C., Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hrițcu, Vilhelm Sjöberg, Andrew Tolmach, and Brent Yorgey. 2018. *Programming Language Foundations*. Software Foundations series, volume 2. Electronic textbook, May.
- Pierce, Benjamin C., and David N. Turner. 2000. “Local Type Inference.” *ACM Trans. Program. Lang. Syst.* (New York, NY, USA) 22, no. 1 (January): 1–44. ISSN: 0164-0925. <https://doi.org/10.1145/345099.345100>. <https://doi.org/10.1145/345099.345100>.
- Plotkin, G. D. 1975. “Call-by-name, call-by-value and the lambda-calculus.” *Theoretical Computer Science* 1, no. 2 (December): 125–159.
- Poletto, Massimiliano, and Vivek Sarkar. 1999. “Linear scan register allocation.” *ACM Trans. Program. Lang. Syst.* 21 (5): 895–913. ISSN: 0164-0925.
- Python Software Foundation. 2021. *The Python Language Reference*. Python Software Foundation, June.
- Reynolds, John C. 1972. “Definitional interpreters for higher-order programming languages.” In *ACM ’72: Proceedings of the ACM Annual Conference*, 717–740. ACM Press.
- Rosen, Kenneth H. 2002. *Discrete Mathematics and Its Applications*. McGraw-Hill Higher Education. ISBN: 0072474777.
- Russell, Stuart J., and Peter Norvig. 2003. *Artificial Intelligence: A Modern Approach*. 2nd ed. Pearson Education. ISBN: 0137903952.
- Sarkar, Dipanwita, Oscar Waddell, and R. Kent Dybvig. 2004. “A nanopass infrastructure for compiler education.” In *ICFP ’04: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, 201–212. ACM Press. ISBN: 1-58113-905-5.
- Shahriyar, Rifat, Stephen M. Blackburn, Xi Yang, and Kathryn M. McKinley. 2013. “Taking Off the Gloves with Reference Counting Immix.” In *OOPSLA ’13: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*. October. <https://doi.org/http://dx.doi.org/10.1145/2509136.2509527>.
- Shidal, Jonathan, Ari J. Spilo, Paul T. Scheid, Ron K. Cytron, and Krishna M. Kavi. 2015. “Recycling Trash in Cache.” In *Proceedings of the 2015 International Symposium on Memory Management*, 118–130. ISMM ’15. New York, NY, USA: ACM. ISBN: 978-1-4503-3589-8. <https://doi.org/10.1145/2754169.2754183>. <http://doi.acm.org/10.1145/2754169.2754183>.
- Shivers, O. 1988. “Control Flow Analysis in Scheme.” In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, 164–174. PLDI ’88. New York, NY, USA: ACM.
- Siebert, Fridtjof. 2001. “Compiler Construction: 10th International Conference, CC 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2–6, 2001 Proceedings.” Chap. Constant-Time Root Scanning for Deterministic Garbage Collection, edited by Reinhard Wilhelm, 304–318. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN: 978-3-540-45306-2. https://doi.org/10.1007/3-540-45306-7_21. http://dx.doi.org/10.1007/3-540-45306-7_21.
- Siek, Jeremy G., and Walid Taha. 2006. “Gradual typing for functional languages.” In *Scheme and Functional Programming Workshop*, 81–92. September.
- Siek, Jeremy G., Peter Thiemann, and Philip Wadler. 2015. “Blame and coercion: Together again for the first time.” In *Conference on Programming Language Design and Implementation*. PLDI. June.
- Sperber, Michael, R. KENT DYBVG, MATTHEW FLATT, ANTON VAN STRAATEN, ROBBY FINDLER, and JACOB MATTHEWS. 2009. “Revised⁶ Report on the Algorithmic Language Scheme.” *Journal of Functional Programming* 19 (August): 1–301. ISSN: 1469-7653. <https://doi.org/10.1017/S0956796809990074>. http://journals.cambridge.org/article_S0956796809990074.

- Steele, Guy L. 1978. *Rabbit: A Compiler for Scheme*. Technical report. Cambridge, MA, USA.
- Steele, Guy L., Jr. 1977. *Data Representations in PDP-10 MacLisp*. AI Memo 420. MIT Artificial Intelligence Lab, September.
- Stroustrup, Bjarne. 1988. "Parameterized Types for C++." In *USENIX C++ Conference*. October.
- Sweigart, Al. 2019. *Automate the Boring Stuff with Python*. No Starch Press.
- Tene, Gil, Balaji Iyengar, and Michael Wolf. 2011. "C4: the continuously concurrent compacting collector." In *Proceedings of the international symposium on Memory management*, 79–88. ISMM '11. New York, NY, USA: ACM. <https://doi.org/http://doi.acm.org/10.1145/1993478.1993491>.
- Tobin-Hochstadt, Sam, and Matthias Felleisen. 2006. "Interlanguage Migration: From Scripts to Programs." In *Dynamic Languages Symposium*.
- Ungar, David. 1984. "Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm." In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, 157–167. SDE 1. New York, NY, USA: ACM. ISBN: 0-89791-131-8. <https://doi.org/10.1145/800020.808261>. <http://doi.acm.org/10.1145/800020.808261>.
- van Wijngaarden, Adriaan. 1966. "Recursive definition of syntax and semantics." In *Formal Language Description Languages for Computer Programming*, edited by Jr. T. B. Steel, 13–24. North-Holland.
- Wadler, Philip, and Robert Bruce Findler. 2009. "Well-typed programs can't be blamed." In *European Symposium on Programming*, 1–16. ESOP. March.
- Weeks, Stephen. 2006. "Whole-Program Compilation in MLton." In *Proceedings of the 2006 Workshop on ML*, 1. ML '06. New York, NY, USA: Association for Computing Machinery. ISBN: 1595934839. <https://doi.org/10.1145/1159876.1159877>. <https://doi.org/10.1145/1159876.1159877>.
- Wilson, Paul. 1992. "Uniprocessor garbage collection techniques." In *Memory Management*, edited by Yves Bekkers and Jacques Cohen, 637:1–42. Lecture Notes in Computer Science. 10.1007/BFb0017182. Springer Berlin / Heidelberg. <http://dx.doi.org/10.1007/BFb0017182>.

Author Index

Subject Index

- abstract syntax tree, 1
- abstract syntax, 1
- administrative normal form, 25
- alias, 94
- alist, 14
- allocate, 93, 104
- ANF, 25
- argument-passing registers, 35
- association list, 14
- AST, 1
- atomic expression, 20

- back-patching, 123
- Backus-Naur Form, 3
- base pointer, 17
- basic block, 19
- BNF, 3
- Boolean, 55
- bottom, 84
- box, 144

- call-live variable, 35
- callee-saved registers, 34
- caller-saved registers, 34
- calling conventions, 34, 47, 126
- Cheney's algorithm, 98
- children, 2
- class, 13
- closure, 140
- closure conversion, 146
- color, 41
- compiler pass, 20
- complex expression, 20
- complex operand, 24
- conclusion, 19, 31, 36, 47, 48, 111, 126
- concrete syntax, 1
- conditional expression, 55
- constant, 3
- contravariant, 184
- control flow, 55
- control-flow graph, 73
- copying collector, 98

- dataflow analysis, 83
- definitional interpreter, 7
- delay, 76

- dictionary, 14
- directed graph, 40
- dynamic typing, 155

- environment, 14

- fixed point, 84
- flat closure, 140
- for/list, 23
- force, 76
- forwarding pointer, 101
- frame, 17, 30, 126, 128
- free variable, 139
- FromSpace, 98
- function, 121
- function application, 121
- function pointer, 121

- generational garbage collector, 119
- generics, 193
- gradual typing, 173
- grammar, 3
- graph, 40
- graph coloring, 41

- heap, 93
- heap allocate, 93

- immediate value, 16
- indirect function call, 126
- indirect jump, 128
- instruction, 17
- instruction selection, 28, 72, 106, 132
- integer, 3
- interfere, 34
- interference graph, 39
- intermediate language, 20
- internal node, 2
- interpreter, 6, 141, 205

- join, 84

- Kleene Fixed-Point Theorem, 84

- lambda, 139
- lattice, 84

- lazy evaluation, 76
- leaf, 2
- least fixed point, 84
- least upper bound, 84
- lexical scoping, 139
- literal, 3
- live objects, 96
- live-after, 37
- live-before, 37
- liveness analysis, 36, 73, 134

- match, 5
- method overriding, 13
- minimum priority queue, 46
- move biasing, 49
- move related, 50
- mutation, 95

- node, 2
- non-terminal, 3

- object, 96
- open recursion, 13

- parameter-passing registers, 35
- parametric polymorphism, 193
- parent, 2
- parse, 1
- partial ordering, 84
- partial evaluation, 9, 31, 70
- pass, 20
- pattern, 5
- pattern matching, 5
- PC, 16
- pointer, 17
- prelude, 18, 31, 36, 47, 48, 102, 111, 126
- priority queue, 46
- procedure call stack, 17, 126
- program, 1
- program counter, 16
- promise, 76

- recursive function, 6
- register, 16
- register allocation, 33, 73, 111, 171
- return address, 17
- root, 2
- root set, 96
- root stack, 101
- runtime system, 28

- saturation, 42
- semantic analysis, 57
- set, 37
- spill, 34
- stack, 17
- stack pointer, 17
- struct, 1, 113
- structural recursion, 6
- structure, 113
- Sudoku, 41

- tagged value, 155
- tail call, 128
- tail position, 22, 27
- terminal, 3
- topological order, 73
- ToSpace, 98
- tuple, 93
- two-space copying collector, 98
- type checking, 57, 141

- unbox, 144
- undirected graph, 40
- unquote-slicing, 95
- unspecified behavior, 8

- variable, 11
- vector, 93

- white-space, 3

- x86, 16, 62, 106, 132, 206