Essentials of Compilation
An Incremental Approach in Python

Essentials of Compilation

An Incremental Approach in Python

Jeremy G. Siek

The MIT Press
Cambridge, Massachusetts
London, England

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

Contents

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. 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 **??** adds support for *objects* and *classes.*
- 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

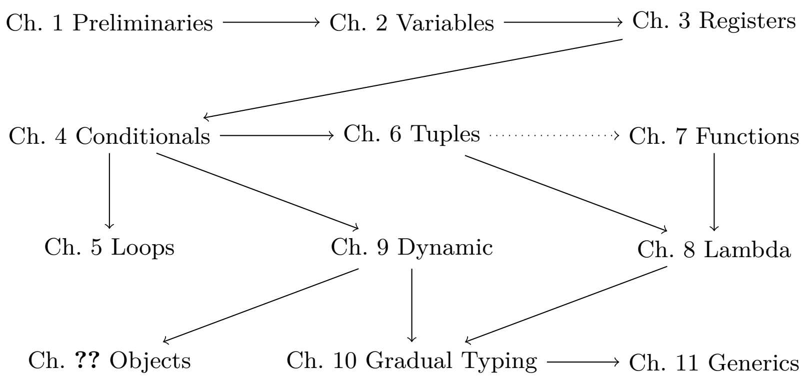


Figure 0.1
Diagram of chapter dependencies.

completing up through Chapter 6 or Chapter 7 and providing some scafolding code to 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. A course that emphasizes object-oriented languages would include Chapter **??**. 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.

This edition of the book uses Python both for the implementation of the compiler and for the input language, so the reader should be proficient with Python. There are many excellent resources for learning Python (Lutz 2013; Barry 2016; Sweigart 2019; Matthes 2019). The support code for this book is in the github repository at the following location:

https://github.com/IUCompilerCourse/

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

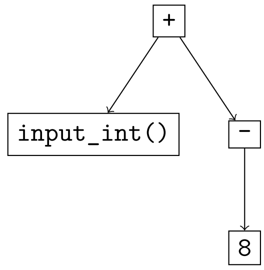
In this chapter we review the basic tools that are needed to implement a compiler. Programs are typically input by a programmer as text, i.e., a sequence of characters. The program-as-text representation is called *concrete syntax*. We use concrete syntax to concisely write down and talk about programs. 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. We use Python's `ast` module to translate 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 Python classes and objects to represent ASTs, especially the classes defined in the standard `ast` module for the Python source language. 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 input 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.

```
input_int() + -8
```



(1.1)

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 use a Python `class` for each kind of node. The following is the class definition for constants.

```
class Constant:
    def __init__(self, value):
        self.value = value
```

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

```
eight = Constant(8)
```

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

The following is the class definition for unary operators.

```
class UnaryOp:
    def __init__(self, op, operand):
        self.op = op
        self.operand = operand
```

The specific operation is specified by the `op` parameter. For example, the class `USub` is for unary subtraction. (More unary operators are introduced in later chapters.) To create an AST that negates the number 8, we write the following.

```
neg_eight = UnaryOp(USub(), eight)
```

The call to the `input_int` function is represented by the `Call` and `Name` classes.

```
class Call:
    def __init__(self, func, args):
        self.func = func
        self.args = args

class Name:
    def __init__(self, id):
        self.id = id
```

To create an AST node that calls `input_int`, we write

```
read = Call(Name('input_int'), [])
```

Finally, to represent the addition in (1.1), we use the `BinOp` class for binary operators.

```
class BinOp:
    def __init__(self, left, op, right):
        self.op = op
        self.left = left
        self.right = right
```

Similar to `UnaryOp`, the specific operation is specified by the `op` parameter, which for now is just an instance of the `Add` class. So to create the AST node that adds negative eight to some user input, we write the following.

```
ast1_1 = BinOp(read, Add(), neg_eight)
```

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. Python 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 $\mathcal{L}_{\mathsf{Int}}$, that consists of integers and arithmetic operations.

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

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

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. In contrast, integers in Python have unlimited precision, but the techniques needed to handle unlimited precision fall outside the scope of this book.

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

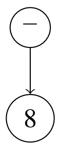
$$\mathit{exp} ::= \texttt{Call(Name('input\_int'),[])} \tag{1.3}$$

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

$$\mathit{exp} ::= \texttt{UnaryOp(USub(),}\mathit{exp}\texttt{)} \tag{1.4}$$

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

```
UnaryOp(USub(), Constant(8))
```



(1.5)

The next grammar rules are for addition and subtraction expressions:

$$\mathit{exp} ::= \texttt{BinOp(}\mathit{exp}\texttt{,Add(),}\mathit{exp}\texttt{)} \tag{1.6}$$

$$\mathit{exp} ::= \texttt{BinOp(Sub(),}\mathit{exp}\texttt{,}\mathit{exp}\texttt{)} \tag{1.7}$$

We can now justify that the AST (1.1) is an *exp* in $\mathcal{L}_{\mathsf{Int}}$. We know that `Call(Name('input_int'),[])` is an *exp* by rule (1.3) and we have already categorized `UnaryOp(USub(), Constant(8))` as an *exp*, so we apply rule (1.6) to show that

```
BinOp(Call(Name('input_int'),[]),Add(),UnaryOp(USub(),Constant(8)))
```

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

If you have an AST for which the above rules do not apply, then the AST is not in $\mathcal{L}_{\mathsf{Int}}$. For example, the program `input_int() * 8` is not in $\mathcal{L}_{\mathsf{Int}}$ 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 language $\mathcal{L}_{\mathsf{Int}}$ includes a second non-terminal *stmt* for statements. There is a statement for printing the value of an expression

$$\mathit{stmt} ::= \texttt{Expr(Call(Name('print'),[}\mathit{exp}\texttt{]))}$$

and a statement that evaluates an expression but ignores the result.

$$\mathit{stmt} ::= \texttt{Expr(}\mathit{exp}\texttt{)}$$

$$
\begin{array}{rcl}
\mathit{exp} & ::= & \mathit{int} \mid \texttt{input\_int()} \mid \texttt{-}\,\mathit{exp} \mid \mathit{exp}\,\texttt{+}\,\mathit{exp} \mid \mathit{exp}\,\texttt{-}\,\mathit{exp} \mid \texttt{(}\mathit{exp}\texttt{)} \\
\mathit{stmt} & ::= & \texttt{print(}\mathit{exp}\texttt{)} \mid \mathit{exp} \\
\mathcal{L}_{\mathsf{Int}} & ::= & \mathit{stmt}^*
\end{array}
$$

Figure 1.1
The concrete syntax of $\mathcal{L}_{\mathsf{Int}}$.

$$
\begin{array}{rcl}
\mathit{binaryop} & ::= & \texttt{Add()} \mid \texttt{Sub()} \\
\mathit{unaryop} & ::= & \texttt{USub()} \\
\mathit{exp} & ::= & \texttt{Constant(}\mathit{int}\texttt{)} \mid \texttt{Call(Name('input\_int'),[])} \\
 & \mid & \texttt{UnaryOp(}\mathit{unaryop}\texttt{,}\mathit{exp}\texttt{)} \mid \texttt{BinOp(}\mathit{binaryop}\texttt{,}\mathit{exp}\texttt{,}\mathit{exp}\texttt{)} \\
\mathit{stmt} & ::= & \texttt{Expr(Call(Name('print'),[}\mathit{exp}\texttt{]))} \mid \texttt{Expr(}\mathit{exp}\texttt{)} \\
\mathcal{L}_{\mathsf{Int}} & ::= & \texttt{Module(}\mathit{stmt}^*\texttt{)}
\end{array}
$$

Figure 1.2
The abstract syntax of $\mathcal{L}_{\mathsf{Int}}$.

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

$$\mathcal{L}_{\mathsf{Int}} ::= \texttt{Module(}\mathit{stmt}^*\texttt{)}$$

The asterisk symbol $*$ indicates a list of the preceding grammar item, in this case, a list of statements. The `Module` class is defined as follows

```
class Module:
    def __init__(self, body):
        self.body = body
```

where `body` is a list of statements.

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 $\mathcal{L}_{\mathsf{Int}}$. 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 $\mathcal{L}_{\mathsf{Int}}$ in Figure 1.2. The concrete syntax for $\mathcal{L}_{\mathsf{Int}}$ is defined in Figure 1.1.

The `parse` function in Python's `ast` module converts the concrete syntax (represented as a string) into an abstract syntax tree.

1.3 Pattern Matching

As mentioned in Section 1.1, compilers often need to access the parts of an AST node. As of version 3.10, Python provides the `match` feature to access the parts of a value. Consider the following example.

```
match ast1_1:
    case BinOp(child1, op, 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 `child1`, `op`, and `child2`, and then prints out the operator. In general, each `case` consists of a *pattern* and a *body*. Patterns are recursively defined to be either a pattern variable, a class name followed by a pattern for each of its constructor's arguments, or other literals such as strings, lists, etc. The body of each `case` may contain arbitrary Python code. The pattern variables can be used in 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 $\mathcal{L}_{\mathsf{Int}}$ 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.

```
def leaf(arith):
    match arith:
        case Constant(n):
            return True
        case Call(Name('input_int'), []):
            return True
        case UnaryOp(USub(), e1):
            return False
        case BinOp(e1, Add(), e2):
            return False
        case BinOp(e1, Sub(), e2):
            return False

print(leaf(Call(Name('input_int'), [])))
print(leaf(UnaryOp(USub(), eight)))
print(leaf(Constant(8)))
```

```
True
False
True
```

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 case for each alternative of that non-terminal and 2) that the pattern in each case corresponds to the corresponding right-hand side of a grammar rule. For the `match` in the `leaf` function, we refer to the grammar for $\mathcal{L}_{\mathsf{Int}}$ in Figure 1.2. The *exp* non-terminal has 4 alternatives, so the `match` has 4 cases. The pattern in each case corresponds to the right-hand side of a grammar rule. For example, the pattern `BinOp(e1, Add(),e2)` corresponds to the right-hand side `BinOp(`*exp*`,Add(),`*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 $\mathcal{L}_{\mathsf{Int}}$. We say that a function is defined by *structural recursion* when it is defined using a sequence of match cases that correspond to a grammar, and the body of each case makes a recursive call on each child node.[1] We define a second function, named `stmt`, that recognizes whether a value is a $\mathcal{L}_{\mathsf{Int}}$ statement. Finally, Figure 1.3 defines `is_Lint`, which determines whether an AST is a program in $\mathcal{L}_{\mathsf{Int}}$. 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 $\mathcal{L}_{\mathsf{Int}}$ 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 Python language is defined in the Python Language Reference (Python Software Foundation 2021) and the CPython interpreter (*CPython github repository* 2021). In this book we use interpreters to specify each language that we 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 $\mathcal{L}_{\mathsf{Int}}$ 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 matches on the `Module` AST node and then invokes `interp_stmt` on each statement in the module. The `interp_stmt` function includes a case for each grammar rule of the *stmt* non-terminal and it calls `interp_exp` on each subexpression. The `interp_exp` function includes a case for each grammar rule of the *exp* non-terminal.

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

```
print(10 + 32)
```

The result is `42`, the answer to life, the universe, and everything: `42`![2] We wrote the above program in concrete syntax whereas the parsed abstract syntax is:

```
Module([Expr(Call(Name('print'), [BinOp(Constant(10), Add(), Constant(32))]))])
```

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

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

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

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

```
def is_exp(e):
  match e:
    case Constant(n):
      return True
    case Call(Name('input_int'), []):
      return True
    case UnaryOp(USub(), e1):
      return is_exp(e1)
    case BinOp(e1, Add(), e2):
      return is_exp(e1) and is_exp(e2)
    case BinOp(e1, Sub(), e2):
      return is_exp(e1) and is_exp(e2)
    case _:
      return False

def stmt(s):
  match s:
    case Expr(Call(Name('print'), [e])):
      return is_exp(e)
    case Expr(e):
      return is_exp(e)
    case _:
      return False

def is_Lint(p):
  match p:
    case Module(body):
      return all([stmt(s) for s in body])
    case _:
      return False

print(is_Lint(Module([Expr(ast1_1)])))
print(is_Lint(Module([Expr(BinOp(read, Sub(),
                UnaryOp(Add(), Constant(8))))])))
```

Figure 1.3
Example of recursive functions for $\mathcal{L}_{\mathsf{Int}}$. These functions recognize whether an AST is in $\mathcal{L}_{\mathsf{Int}}$.

What is the result of the above program?

Moving on to the last feature of the $\mathcal{L}_{\mathsf{Int}}$ language, the `input_int` 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(Module([Expr(Call(Name('print'), [ast1_1]))]))
```

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

```
def interp_exp(e):
    match e:
        case BinOp(left, Add(), right):
            l = interp_exp(left); r = interp_exp(right)
            return l + r
        case BinOp(left, Sub(), right):
            l = interp_exp(left); r = interp_exp(right)
            return l - r
        case UnaryOp(USub(), v):
            return - interp_exp(v)
        case Constant(value):
            return value
        case Call(Name('input_int'), []):
            return int(input())

def interp_stmt(s):
    match s:
        case Expr(Call(Name('print'), [arg])):
            print(interp_exp(arg))
        case Expr(value):
            interp_exp(value)

def interp_Lint(p):
    match p:
        case Module(body):
            for s in body:
                interp_stmt(s)
```

Figure 1.4
Interpreter for the $\mathcal{L}_{\mathsf{Int}}$ language.

We include the `input_int` operation in $\mathcal{L}_{\mathsf{Int}}$ so a clever student cannot implement a compiler for $\mathcal{L}_{\mathsf{Int}}$ that simply runs the interpreter during compilation to obtain the output and then generates the trivial code to produce the output.[3]

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

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

P_2 on their respective interpreters with same input i yields the same output o.

$$
\begin{tikzcd}
P_1 \arrow[r, "\text{compile}"] \arrow[dr, "\text{interp\_}\mathcal{L}_1(i)"'] & P_2 \arrow[d, "\text{interp\_}\mathcal{L}_2(i)"] \\
 & o
\end{tikzcd}
\tag{1.8}
$$

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 $\mathcal{L}_{\mathsf{Int}}$ programs into $\mathcal{L}_{\mathsf{Int}}$ 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

```
print(input_int() + -(5 + 3) )
```

our compiler translates it into the program

```
print(input_int() + -8)
```

Figure 1.5 gives the code for a simple partial evaluator for the $\mathcal{L}_{\mathsf{Int}}$ language. The output of the partial evaluator is a program in $\mathcal{L}_{\mathsf{Int}}$. 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, we can test whether it satisfies Diagram 1.8.

Exercise 1 Create three programs in the $\mathcal{L}_{\mathsf{Int}}$ 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`.

```
def pe_neg(r):
  match r:
    case Constant(n):
      return Constant(-n)
    case _:
      return UnaryOp(USub(), r)

def pe_add(r1, r2):
  match (r1, r2):
    case (Constant(n1), Constant(n2)):
      return Constant(n1 + n2)
    case _:
      return BinOp(r1, Add(), r2)

def pe_sub(r1, r2):
  match (r1, r2):
    case (Constant(n1), Constant(n2)):
      return Constant(n1 - n2)
    case _:
      return BinOp(r1, Sub(), r2)

def pe_exp(e):
  match e:
    case BinOp(left, Add(), right):
      return pe_add(pe_exp(left), pe_exp(right))
    case BinOp(left, Sub(), right):
      return pe_sub(pe_exp(left), pe_exp(right))
    case UnaryOp(USub(), v):
      return pe_neg(pe_exp(v))
    case Constant(value):
      return e
    case Call(Name('input_int'), []):
      return e

def pe_stmt(s):
  match s:
    case Expr(Call(Name('print'), [arg])):
      return Expr(Call(Name('print'), [pe_exp(arg)]))
    case Expr(value):
      return Expr(pe_exp(value))

def pe_P_int(p):
  match p:
    case Module(body):
      new_body = [pe_stmt(s) for s in body]
      return Module(new_body)
```

Figure 1.5
A partial evaluator for $\mathcal{L}_{\mathsf{Int}}$.

2 Integers and Variables

This chapter is about compiling a subset of Python to x86-64 assembly code (Intel 2015). The subset, named $\mathcal{L}_{\mathsf{Var}}$, includes integer arithmetic and local variables. We often refer to x86-64 simply as x86. The chapter begins with a description of the $\mathcal{L}_{\mathsf{Var}}$ 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 $\mathcal{L}_{\mathsf{Var}}$. We introduce more x86 instructions in later chapters. After introducing $\mathcal{L}_{\mathsf{Var}}$ and x86, we reflect on their differences and come up with a plan to break down the translation from $\mathcal{L}_{\mathsf{Var}}$ 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 $\mathcal{L}_{\mathsf{Var}}$ to x86 in a short time. To give the reader a feeling for the scale of this first compiler, the instructor solution for the $\mathcal{L}_{\mathsf{Var}}$ compiler is approximately 300 lines of code.

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

The $\mathcal{L}_{\mathsf{Var}}$ language extends the $\mathcal{L}_{\mathsf{Int}}$ language with variables. The concrete syntax of the $\mathcal{L}_{\mathsf{Var}}$ 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 Python identifier. As in $\mathcal{L}_{\mathsf{Int}}$, `input_int` is a nullary operator, `-` is a unary operator, and `+` is a binary operator. Similar to $\mathcal{L}_{\mathsf{Int}}$, the abstract syntax of $\mathcal{L}_{\mathsf{Var}}$ includes the `Module` instance to mark the top of the program. Despite the simplicity of the $\mathcal{L}_{\mathsf{Var}}$ language, it is rich enough to exhibit several compilation techniques.

The $\mathcal{L}_{\mathsf{Var}}$ language includes assignment statements, which define a variable for use in later statements and initializes the variable with the value of an expression. The abstract syntax for assignment is defined in Figure 2.2. The concrete syntax for assignment is

var `=` *exp*

For example, the following program initializes the variable `x` to 32 and then prints the result of `10 + x`, producing 42.

```
x = 12 + 20
print(10 + x)
```

$$
\begin{array}{rcl}
\mathit{exp} & ::= & \mathit{int} \mid \texttt{input\_int()} \mid \texttt{-}\mathit{exp} \mid \mathit{exp}\,\texttt{+}\,\mathit{exp} \mid \mathit{exp}\,\texttt{-}\,\mathit{exp} \mid \texttt{(}\mathit{exp}\texttt{)} \\
\mathit{stmt} & ::= & \texttt{print(}\mathit{exp}\texttt{)} \mid \mathit{exp} \\
\hline
\mathit{exp} & ::= & \mathit{var} \\
\mathit{stmt} & ::= & \mathit{var}\,\texttt{=}\,\mathit{exp} \\
\mathcal{L}_{\mathsf{Var}} & ::= & \mathit{stmt}^*
\end{array}
$$

Figure 2.1
The concrete syntax of $\mathcal{L}_{\mathsf{Var}}$.

$$
\begin{array}{rcl}
\mathit{binaryop} & ::= & \texttt{Add()} \mid \texttt{Sub()} \\
\mathit{unaryop} & ::= & \texttt{USub()} \\
\mathit{exp} & ::= & \texttt{Constant(}\mathit{int}\texttt{)} \mid \texttt{Call(Name('input\_int'),[])} \\
 & \mid & \texttt{UnaryOp(}\mathit{unaryop}\texttt{,}\mathit{exp}\texttt{)} \mid \texttt{BinOp(}\mathit{binaryop}\texttt{,}\mathit{exp}\texttt{,}\mathit{exp}\texttt{)} \\
\mathit{stmt} & ::= & \texttt{Expr(Call(Name('print'),[}\mathit{exp}\texttt{]))} \mid \texttt{Expr(}\mathit{exp}\texttt{)} \\
\hline
\mathit{exp} & ::= & \texttt{Name(}\mathit{var}\texttt{)} \\
\mathit{stmt} & ::= & \texttt{Assign([Name(}\mathit{var}\texttt{)], }\mathit{exp}\texttt{)} \\
\mathcal{L}_{\mathsf{Var}} & ::= & \texttt{Module(}\mathit{stmt}^*\texttt{)}
\end{array}
$$

Figure 2.2
The abstract syntax of $\mathcal{L}_{\mathsf{Var}}$.

2.1.1 Extensible Interpreters via Method Overriding

To prepare for discussing the interpreter of $\mathcal{L}_{\mathsf{Var}}$, 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 $\mathcal{L}_{\mathsf{Var}}$ would handle the case for variables but dispatch to an interpreter for $\mathcal{L}_{\mathsf{Int}}$ in the rest of the cases. The following code sketches this idea. (We explain the `env` parameter soon, in Section 2.1.2.)

```
def interp_Lint(e, env):
  match e:
    case UnaryOp(USub(), e1):
       return - interp_Lint(e1, env)
    ...
```

```
def interp_Lvar(e, env):
  match e:
    case Name(id):
      return env[id]
    case _:
      return interp_Lint(e, env)
```

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

```
y = 10
print(-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 $\mathcal{L}_{\mathsf{Int}}$ and $\mathcal{L}_{\mathsf{Var}}$ using a Python `class` definition. We define one class for each language and define a method for interpreting expressions inside each class. The class for $\mathcal{L}_{\mathsf{Var}}$ inherits from the class for $\mathcal{L}_{\mathsf{Int}}$ and the method `interp_exp` in $\mathcal{L}_{\mathsf{Var}}$ overrides the `interp_exp` in $\mathcal{L}_{\mathsf{Int}}$. Note that the default case of `interp_exp` in $\mathcal{L}_{\mathsf{Var}}$ uses `super` to invoke `interp_exp`, and because $\mathcal{L}_{\mathsf{Var}}$ inherits from $\mathcal{L}_{\mathsf{Int}}$, that dispatches to the `interp_exp` in $\mathcal{L}_{\mathsf{Int}}$.

```
class InterpLint:
  def interp_exp(e):
    match e:
      case UnaryOp(USub(), e1):
        return -self.interp_exp(e1)
      ...
  ...
```

```
def InterpLvar(InterpLint):
  def interp_exp(e):
    match e:
      case Name(id):
        return env[id]
      case _:
        return super().interp_exp(e)
  ...
```

Getting back to the troublesome example, repeated here:

```
y = 10
print(-y)
```

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

```
InterpLvar().interp_exp(e0)
```

To process the `-` operator, the default case of `interp_exp` in $\mathcal{L}_{\mathsf{Var}}$ dispatches to the `interp_exp` method in $\mathcal{L}_{\mathsf{Int}}$. But then for the recursive method call, it dispatches back to `interp_exp` in $\mathcal{L}_{\mathsf{Var}}$, 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 $\mathcal{L}_{\mathsf{Var}}$

Having justified the use of classes and methods to implement interpreters, we revisit the definitional interpreter for $\mathcal{L}_{\mathsf{Int}}$ in Figure 2.3 and then extend it to create an interpreter for $\mathcal{L}_{\mathsf{Var}}$ in Figure 2.4. The interpreter for $\mathcal{L}_{\mathsf{Var}}$ adds two new `match` cases for variables and assignment. For assignment 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 a Python dictionary to represent the environment. 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

```
class InterpLint:
  def interp_exp(self, e, env):
    match e:
      case BinOp(left, Add(), right):
        return self.interp_exp(left, env) + self.interp_exp(right, env)
      case BinOp(left, Sub(), right):
        return self.interp_exp(left, env) - self.interp_exp(right, env)
      case UnaryOp(USub(), v):
        return - self.interp_exp(v, env)
      case Constant(value):
        return value
      case Call(Name('input_int'), []):
        return int(input())

  def interp_stmts(self, ss, env):
    if len(ss) == 0:
      return
    match ss[0]:
      case Expr(Call(Name('print'), [arg])):
        print(self.interp_exp(arg, env), end='')
        return self.interp_stmts(ss[1:], env)
      case Expr(value):
        self.interp_exp(value, env)
        return self.interp_stmts(ss[1:], env)

  def interp(self, p):
    match p:
      case Module(body):
        self.interp_stmts(body, {})

def interp_Lint(p):
  return InterpLint().interp(p)
```

Figure 2.3
Interpreter for $\mathcal{L}_{\mathsf{Int}}$ as a class.

interpreter encounters an assignment, it evaluates the initializing expression and then associates the resulting value with the variable in the environment.

The goal for this chapter is to implement a compiler that translates any program P_1 written in the $\mathcal{L}_{\mathsf{Var}}$ 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 criteria in the following diagram.

$$
\begin{tikzcd}
P_1 \arrow[r, "\text{compile}"] \arrow[dr, "\texttt{interp\_Lvar}"'] & P_2 \arrow[d, "\texttt{interp\_x86int}"] \\
 & n
\end{tikzcd}
$$

```
class InterpLvar(InterpLint):
  def interp_exp(self, e, env):
    match e:
      case Name(id):
        return env[id]
      case _:
        return super().interp_exp(e, env)

  def interp_stmts(self, ss, env):
    if len(ss) == 0:
      return
    match ss[0]:
      case Assign([lhs], value):
        env[lhs.id] = self.interp_exp(value, env)
        return self.interp_stmts(ss[1:], env)
      case _:
        return super().interp_stmts(ss, env)

def interp_Lvar(p):
  return InterpLvar().interp(p)
```

Figure 2.4
Interpreter for the $\mathcal{L}_{\mathsf{Var}}$ language.

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

2.2 The $\mathtt{x86}_{\mathsf{Int}}$ Assembly Language

Figure 2.5 defines the concrete syntax for $\mathtt{x86}_{\mathsf{Int}}$. 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 2.5. 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.

$$
\begin{array}{lcl}
\mathit{reg} & ::= & \texttt{rsp} \mid \texttt{rbp} \mid \texttt{rax} \mid \texttt{rbx} \mid \texttt{rcx} \mid \texttt{rdx} \mid \texttt{rsi} \mid \texttt{rdi} \mid \\
 & & \texttt{r8} \mid \texttt{r9} \mid \texttt{r10} \mid \texttt{r11} \mid \texttt{r12} \mid \texttt{r13} \mid \texttt{r14} \mid \texttt{r15} \\
\mathit{arg} & ::= & \$\mathit{int} \mid \%\mathit{reg} \mid \mathit{int}(\%\mathit{reg}) \\
\mathit{instr} & ::= & \texttt{addq}\,\mathit{arg},\mathit{arg} \mid \texttt{subq}\,\mathit{arg},\mathit{arg} \mid \texttt{negq}\,\mathit{arg} \mid \texttt{movq}\,\mathit{arg},\mathit{arg} \mid \\
 & & \texttt{callq}\,\mathit{label} \mid \texttt{pushq}\,\mathit{arg} \mid \texttt{popq}\,\mathit{arg} \mid \texttt{retq} \\
\mathrm{x86}_{\mathsf{Int}} & ::= & \texttt{.globl main} \\
 & & \texttt{main:}\ \mathit{instr}^{*}
\end{array}
$$

Figure 2.5
The syntax of the $\mathrm{x86}_{\mathsf{Int}}$ assembly language (AT&T syntax).

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

Figure 2.6
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.

Appendix A.1 contains a quick-reference for all of the x86 instructions used in this book.

Figure 2.6 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.

We exhibit the use of memory for storing intermediate results in the next example. Figure 2.7 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 2.8. 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.

```
        .globl main
main:
        pushq  %rbp
        movq   %rsp, %rbp
        subq   $16, %rsp
        movq   $10, -8(%rbp)
        negq   -8(%rbp)
        movq   -8(%rbp), %rax
        addq   $52, %rax
        addq   $16, %rsp
        popq   %rbp
        retq
```

Figure 2.7
An x86 program that computes 52 + -10.

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

Figure 2.8
Memory layout of a frame.

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 2.8 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 2.7, 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 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

reg	::=	`'rsp'` | `'rbp'` | `'rax'` | `'rbx'` | `'rcx'` | `'rdx'` | `'rsi'` | `'rdi'` | `'r8'` | `'r9'` | `'r10'` | `'r11'` | `'r12'` | `'r13'` | `'r14'` | `'r15'`
arg	::=	`Immediate(`*int*`)` | `Reg(`*reg*`)` | `Deref(`*reg*`,`*int*`)`
instr	::=	`Instr('addq',[`*arg*`,`*arg*`])` | `Instr('subq',[`*arg*`,`*arg*`])`
	|	`Instr('movq',[`*arg*`,`*arg*`])` | `Instr('negq',[`*arg*`])`
	|	`Instr('pushq',[`*arg*`])` | `Instr('popq',[`*arg*`])`
	|	`Callq(`*label*`,`*int*`)` | `Retq()` | `Jump(`*label*`)`
$\mathrm{x86}_{\mathsf{Int}}$	::=	`X86Program(`$instr^*$`)`

Figure 2.9
The abstract syntax of $\mathrm{x86}_{\mathsf{Int}}$ assembly.

(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 first instruction after the prelude is `movq $10, -8(%rbp)`, which stores 10 in variable 1. The instruction `negq -8(%rbp)` changes the contents of variable 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 *conclusion* of the `main` function consists of the last three instructions. 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 2.9. We refer to this language as $\mathrm{x86}_{\mathsf{Int}}$. The main difference compared to the concrete syntax of $\mathrm{x86}_{\mathsf{Int}}$ (Figure 2.5) is that labels, instruction names, and register names are explicitly represented by strings. 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 $\mathcal{L}_{\mathsf{Var}}$ 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, $\mathcal{L}_{\mathsf{Var}}$ 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 $\mathcal{L}_{\mathsf{Var}}$ operator can be a deeply-nested expression, whereas x86 instructions restrict their arguments to be integer constants, registers, and memory locations.
3. A program in $\mathcal{L}_{\mathsf{Var}}$ can have any number of variables whereas x86 has 16 registers and the procedure call stack.

We ease the challenge of compiling from $\mathcal{L}_{\mathsf{Var}}$ 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 $\mathcal{L}_{\mathsf{Var}}$ as its input language and the last pass has $\mathrm{x86}_{\mathsf{Int}}$ as its output language. In between we can choose whichever language is most convenient for expressing the output of each pass, whether that be $\mathcal{L}_{\mathsf{Var}}$, $\mathrm{x86}_{\mathsf{Int}}$, 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 $\mathcal{L}_{\mathsf{Var}}$ consists of the following passes.

`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.

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

`assign_homes` replaces variables with registers or stack locations.

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.

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 iteratively 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

$$
\begin{tikzcd}
\mathcal{L}_{\mathsf{Var}} \arrow[r, bend left, "\texttt{remove\_complex.}"] & \mathcal{L}^{mon}_{\mathsf{Var}} \arrow[d, bend right, "\texttt{select\_instr.}"'] & & & \\
& \text{x86}_{\mathsf{Var}} \arrow[r, bend right, "\texttt{assign\_homes}"'] & \text{x86}_{\mathsf{Var}} \arrow[r, bend left, "\texttt{patch\_instr.}"] & \text{x86}_{\mathsf{Int}} \arrow[r, bend right, "\texttt{prelude\_and\_concl.}"'] & \text{x86}_{\mathsf{Int}}
\end{tikzcd}
$$

Figure 2.10
Diagram of the passes for compiling $\mathcal{L}_{\mathsf{Var}}$.

pass named `patch_instructions` that uses a reserved register to fix outstanding problems.

Figure 2.10 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 $\text{x86}_{\mathsf{Var}}$ language, which extends $\text{x86}_{\mathsf{Int}}$ 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. The remainder of this chapter provides guidance on the implementation of each of the compiler passes in Figure 2.10.

2.4 Remove Complex Operands

The `remove_complex_operands` pass compiles $\mathcal{L}_{\mathsf{Var}}$ 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.

```
x = 42 + -10
print(x + 10)
```

$\Rightarrow$

```
tmp_0 = -10
x = 42 + tmp_0
tmp_1 = x + 10
print(tmp_1)
```

Figure 2.11 presents the grammar for the output of this pass, the language $\mathcal{L}^{mon}_{\mathsf{Var}}$. 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 `Call(Name('input_int'),[])`. A language with this separation between pure versus side-effecting expressions is said to be in monadic normal form (Moggi 1991;

$$
\begin{array}{rcl}
\mathit{atm} & ::= & \texttt{Constant(}\mathit{int}\texttt{)} \mid \texttt{Name(}\mathit{var}\texttt{)} \\
\mathit{exp} & ::= & \mathit{atm} \mid \texttt{Call(Name('input\_int'),[])} \\
 & \mid & \texttt{UnaryOp(}\mathit{unaryop}\texttt{,}\mathit{atm}\texttt{)} \mid \texttt{BinOp(}\mathit{atm}\texttt{,}\mathit{binaryop}\texttt{,}\mathit{atm}\texttt{)} \\
\mathit{stmt} & ::= & \texttt{Expr(Call(Name('print'),[}\mathit{atm}\texttt{]))} \mid \texttt{Expr(}\mathit{exp}\texttt{)} \\
 & \mid & \texttt{Assign([Name(}\mathit{var}\texttt{)], }\mathit{exp}\texttt{)} \\
\mathcal{L}^{mon}_{\mathsf{Var}} & ::= & \texttt{Module(}\mathit{stmt}^*\texttt{)}
\end{array}
$$

Figure 2.11
$\mathcal{L}^{mon}_{\mathsf{Var}}$ is $\mathcal{L}_{\mathsf{Var}}$ with operands restricted to atomic expressions.

Danvy 2003) which explains the *mon* in the name $\mathcal{L}^{mon}_{\mathsf{Var}}$. 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 behaviour 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 $\mathcal{L}^{mon}_{\mathsf{Var}}$ 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 an auxiliary method named `rco_exp` with two parameters: an $\mathcal{L}_{\mathsf{Var}}$ expression and a Boolean that specifies whether the expression needs to become atomic or not. The `rco_exp` method should return a pair consisting of the new expression and a list of pairs, associating new temporary variables with their initializing expressions.

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

```
-10
```
⇒
```
tmp_1
[(tmp_1, -10)]
```

Take special care of programs such as the following that assign an atomic expression to a variable. You should leave such assignments unchanged, as shown in the program on the right

```
a = 42
b = a
print(b)
```
⇒
```
a = 42
b = a
print(b)
```

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

```
tmp_1 = 42
a = tmp_1
tmp_2 = a
b = tmp_2
print(b)
```

Exercise 2 Implement the `remove_complex_operands` pass in `compiler.py`, creating auxiliary functions for each non-terminal in the grammar, i.e., `rco_exp` and `rco_stmt`. We recommend you use the function `utils.generate_name()` to generate fresh names from a stub string.

Exercise 3 Create five $\mathcal{L}_{\mathsf{Var}}$ programs that exercise the most interesting parts of the `remove_complex_operands` pass. 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 `.py`. Run the `run-tests.py` script in the support code to check whether the output programs produce the same result as the input programs.

2.5 Select Instructions

In the `select_instructions` pass we begin the work of translating to $\text{x86}_{\mathsf{Var}}$. The target language of this pass is a variant of x86 that still uses variables, so we add an AST node of the form `Name(`*var*`)` to the *arg* non-terminal of the $\text{x86}_{\mathsf{Int}}$ abstract syntax (Figure 2.9). We recommend implementing an auxiliary function named `select_stmt` for the *stmt* non-terminal.

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.

$$var \texttt{ = } atm_1 \texttt{ + } atm_2 \quad \Rightarrow \quad \begin{array}{l}\texttt{movq } arg_1\texttt{, } var\\ \texttt{addq } arg_2\texttt{, } var\end{array}$$

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.

$$var \texttt{ = } atm_1 \texttt{ + } var \quad \Rightarrow \quad \texttt{addq } arg_1\texttt{, } var$$

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

Similarly, we translate the `print` operation, shown below, into a call to the `print_int` function defined in `runtime.c`. In x86, the first six arguments to functions are passed in registers, with the first argument passed in register `rdi`. So we move the *arg* into `rdi` and then call `print_int` using the `callq` instruction.

```
print(atm)                  ⇒    movq arg, %rdi
                                 callq print_int
```

We recommend that you use the function `utils.label_name()` to transform a string into an label argument suitably suitable for, e.g., the target of the `callq` instruction. This practice makes your compiler portable across Linus and Mac OS X, which requires an underscore prefixed to all labels.

Exercise 4 Implement the `select_instructions` pass in `compiler.py`. Create three new example programs that are designed to exercise all of the interesting cases in this pass. Run the `run-tests.py` script to to check whether the output programs produce the same result as the input programs.

2.6 Assign Homes

The `assign_homes` pass compiles $\text{x86}_{\mathsf{Var}}$ programs to $\text{x86}_{\mathsf{Var}}$ 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 $\mathcal{L}_{\mathsf{Var}}$ program from Section 2.4.

```
a = 42
b = a
print(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 uses of variables with stack locations. 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 field `stack_space` of the `X86Program` node, 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 5 Implement the `assign_homes` pass in `compiler.py`, defining auxiliary functions for each of the non-terminals in the $\mathtt{x86}_{\mathsf{Var}}$ grammar. We recommend that the auxiliary functions take an extra parameter that maps variable names to homes (stack locations for now). Run the `run-tests.py` script to to check whether the output programs produce the same result as the input programs.

2.7 Patch Instructions

The `patch_instructions` pass compiles from $\mathtt{x86}_{\mathsf{Var}}$ to $\mathtt{x86}_{\mathsf{Int}}$ 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.

```
a = 42
b = a
print(b)
```

The `assign_homes` pass produces the following translation.

```
movq 42, -8(%rbp)
movq -8(%rbp), -16(%rbp)
movq -16(%rbp), %rdi
callq print_int
```

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

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

Exercise 6 Implement the `patch_instructions` pass in `compiler.py`. Create three new example programs that are designed to exercise all of the interesting cases in this pass. Run the `run-tests.py` script to to check whether the output programs produce the same result as the input programs.

2.8 Generate Prelude and Conclusion

The last step of the compiler from $\mathcal{L}_{\mathsf{Var}}$ to x86 is to generate the `main` function with a prelude and conclusion wrapped around the rest of the program, as shown in Figure 2.7 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 Python `platform` library includes a `system()` function that returns `'Linux'`, `'Windows'`, or `'Darwin'` (for Mac).

Exercise 7 Implement the `prelude_and_conclusion` pass in `compiler.py`. Run the `run-tests.py` script to to check whether the output programs produce the same result as the input programs. That script translates the x86 AST that you produce into a string by invoking the `repr` method that is implemented by the x86 AST classes in `x86_ast.py`.

2.9 Challenge: Partial Evaluator for $\mathcal{L}_{\mathsf{Var}}$

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

Exercise 8 Adapt the partial evaluator from Section 1.6 (Figure 1.5) so that it applies to $\mathcal{L}_{\mathsf{Var}}$ programs instead of $\mathcal{L}_{\mathsf{Int}}$ programs. Recall that $\mathcal{L}_{\mathsf{Var}}$ adds variables and assignment to the $\mathcal{L}_{\mathsf{Int}}$ language, so you will need to add cases for them in the `pe_exp` and `pe_stmt` functions. 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 9 Improve on the partial evaluator by replacing the `pe_neg` and `pe_add` auxiliary functions with functions that know more about arithmetic. For example, your partial evaluator should translate

`1 + (input_int() + 1)` into `2 + input_int()`

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.

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

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 $\mathcal{L}_{\mathsf{Var}}$ 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.

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

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

```
v = 1
w = 42
x = v + 7
y = x
z = x + w
print(z + (- y))
```

After instruction selection:

```
movq $1, v
movq $42, w
movq v, x
addq $7, x
movq x, y
movq x, z
addq w, z
movq y, tmp_0
negq tmp_0
movq z, tmp_1
addq tmp_0, tmp_1
movq tmp_1, %rdi
callq print_int
```

Figure 3.1
A running example for register allocation.

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 $\mathcal{L}_{\mathsf{Var}}$ 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 functions we care about are `read_int` and `print_int`, which take zero and one argument, respectively. 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 $\mathcal{L}_{\mathsf{Var}}$ 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 `input_int`. The variable `x` is call-live because it is in-use during the second call to `input_int`; we must ensure that the value in `x` does not get overwritten during the call to `input_int`. 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 `input_int`. 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 that variable `y` is assigned to `rcx`, a caller-saved register, because `y` is not a call-live variable.

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

```
x = input_int()
y = input_int()
print((x + y) + 42)
```

Generated x86 assembly:

```
        .globl main
main:
        pushq %rbp
        movq %rsp, %rbp
        pushq %rbx
        subq $8, %rsp
        callq read_int
        movq %rax, %rbx
        callq read_int
        movq %rax, %rcx
        movq %rbx, %rdx
        addq %rcx, %rdx
        movq %rdx, %rcx
        addq $42, %rcx
        movq %rcx, %rdi
        callq print_int
        addq $8, %rsp
        popq %rbx
        popq %rbp
        retq
```

Figure 3.2
An example with function calls.

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` function 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?

```
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 $I_1, \ldots, I_n$ be the instruction sequence. We write $L_{\mathsf{after}}(k)$ for the set of live locations after instruction I_k and $L_{\mathsf{before}}(k)$ for the set of live locations before instruction I_k. We recommend representing these sets with the Python `set` data structure.

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

$$L_{\mathsf{after}}(k) = L_{\mathsf{before}}(k+1) \tag{3.9}$$

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

$$L_{\mathsf{after}}(n) = \emptyset \tag{3.10}$$

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

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

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

Let us walk through the above example, applying these formulas starting with the instruction on line 5. We collect the answers in Figure 3.3. 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 $\{\mathtt{b}, \mathtt{c}\}$ because it reads from variables `b` and `c` (formula 3.11), that is

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

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

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

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

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

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

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

$$L_{\text{before}}(1) = \emptyset, L_{\text{after}}(1) = \{\texttt{a}\}$$

$$L_{\text{before}}(2) = \{\texttt{a}\}, L_{\text{after}}(2) = \{\texttt{a}\}$$

$$L_{\text{before}}(3) = \{\texttt{a}\}, L_{\text{after}}(2) = \{\texttt{c}\}$$

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

$$L_{\text{before}}(5) = \{\texttt{b}, \texttt{c}\}, L_{\text{after}}(5) = \emptyset$$

Figure 3.3
Example output of liveness analysis on a short example.

```
movq $1, v
                   {v}
movq $42, w
                   {w, v}
movq v, x
                   {w, x}
addq $7, x
                   {w, x}
movq x, y
                   {w, x, y}
movq x, z
                   {w, y, z}
addq w, z
                   {y, z}
movq y, tmp_0
                   {tmp_0, z}
negq tmp_0
                   {tmp_0, z}
movq z, tmp_1
                   {tmp_0, tmp_1}
addq tmp_0, tmp_1
                   {tmp_1}
movq tmp_1, %rdi
                   {rdi}
callq print_int
                   {}
```

Figure 3.4
The running example annotated with live-after sets.

Exercise 10 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 3.4.

Exercise 11 Implement the `uncover_live` function. Return a dictionary that maps each instruction to its live-after set. 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 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 provide implementations of directed and undirected graph data structures in the file `graph.py` of the support code.

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 (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 $v \in L_{\mathsf{after}}(k)$, 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 $v \in L_{\mathsf{after}}(k)$, if $v \neq d$, add the edge (d, v).

Working from the top to bottom of Figure 3.4, 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}`. Rule 1 applies but there is no interference because `v` is the destination of the move. The fourth instruction is `addq $7, x` and the live-after set is `{w, x}`. Rule 2 applies so `x` interferes with `w`. The next instruction

`movq $1, v`	no interference
`movq $42, w`	`w` interferes with `v`
`movq v, x`	`x` interferes with `w`
`addq $7, x`	`x` interferes with `w`
`movq x, y`	`y` interferes with `w` but not `x`
`movq x, z`	`z` interferes with `w` and `y`
`addq w, z`	`z` interferes with `y`
`movq y, tmp_0`	`tmp_0` interferes with `z`
`negq tmp_0`	`tmp_0` interferes with `z`
`movq z, tmp_1`	`tmp_0` interferes with `tmp_1`
`addq tmp_0, tmp_1`	no interference
`movq tmp_1, %rdi`	no interference
`callq print_int`	no interference.

Figure 3.5
Interference results for the running example.

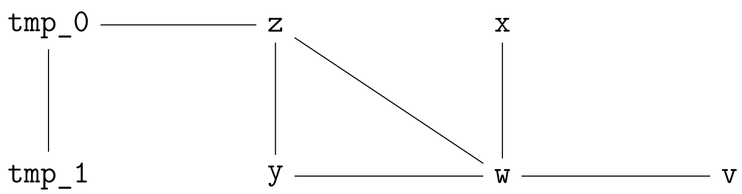


Figure 3.6
The interference graph of the example program.

is `movq x, y` and the live-after set is {`w`, `x`, `y`}. Rule 1 applies, so `y` interferes with `w` but not `x` because `x` is the source of the move and therefore `x` and `y` hold the same value. Figure 3.5 lists the interference results for all of the instructions and the resulting interference graph is shown in Figure 3.6.

Exercise 12 Implement a function named `build_interference` according to the algorithm suggested above that returns the interference graph.

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.

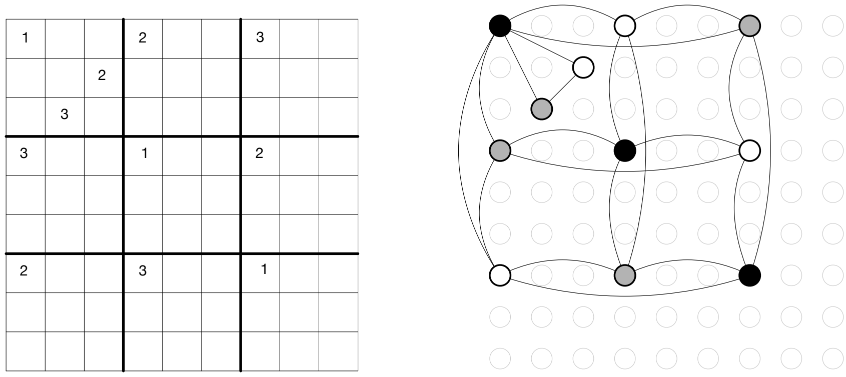


Figure 3.7
A Sudoku game board and the corresponding colored graph.

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

If you can color the remaining vertices in the graph with the nine colors, then you have also solved the corresponding game of Sudoku. Figure 3.7 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:

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

where $\mathsf{adjacent}(u)$ 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,

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

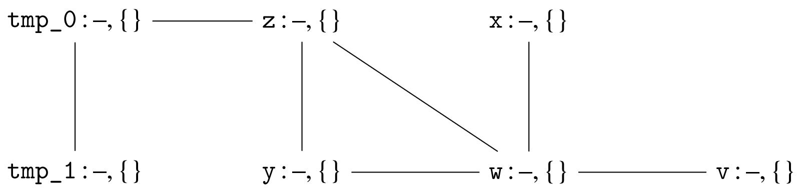
$W \leftarrow \mathrm{vertices}(G)$
while $W \neq \emptyset$ do
    pick a vertex u from W with the highest saturation,
        breaking ties randomly
    find the lowest color c that is not in $\{\mathrm{color}[v] : v \in \mathrm{adjacent}(u)\}$
    $\mathrm{color}[u] \leftarrow c$
    $W \leftarrow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

Figure 3.8
The saturation-based greedy graph coloring algorithm.

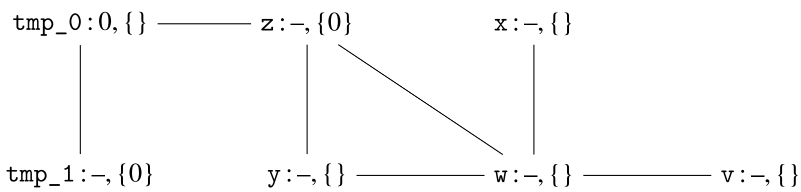
great. If not, backtrack and choose the next possibility. One good thing about Pencil Marks is that it reduces the degree of branching in the search tree. Nevertheless, backtracking can be terribly time consuming. One way to reduce the amount of backtracking is to use the most-constrained-first heuristic (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 3.8 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`.

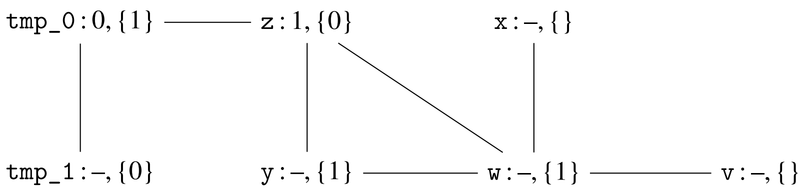
With the DSATUR algorithm in hand, let us return to the running example and consider how to color the interference graph in Figure 3.6. We annotate each variable node with a dash to indicate that it has not yet been assigned a color. The saturation sets are also shown for each node; all of them start as the empty set. (We do not include the register nodes in the graph below because there were no interference edges involving registers in this program, but in general there can be.)



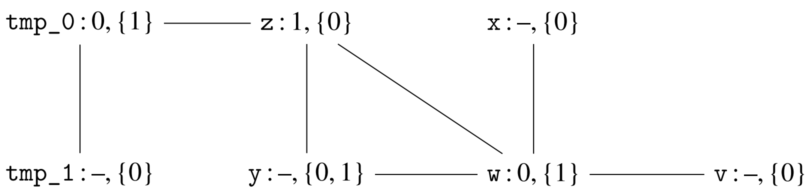
The algorithm says to select a maximally saturated vertex, but they are all equally saturated. So we flip a coin and pick `tmp_0` then color it with the first available integer, which is 0. We mark 0 as no longer available for `tmp_1` and `z` because they interfere with `tmp_0`.



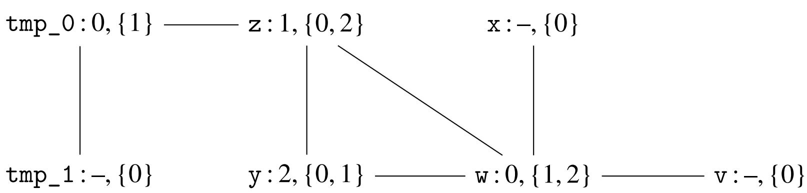
We repeat the process. The most saturated vertices are `z` and `tmp_1`, so we choose `z` and color it with the first available number, which is 1. We add 1 to the saturation for the neighboring vertices `tmp_0`, `y`, and `w`.



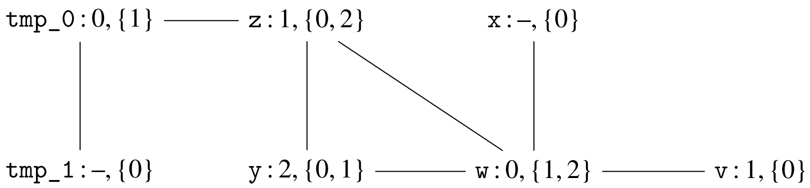
The most saturated vertices are now `tmp_1`, `w`, and `y`. We color `w` with the first available color, which is 0.



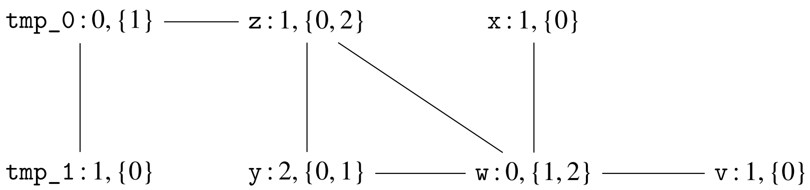
Now `y` is the most saturated, so we color it with 2.



The most saturated vertices are `tmp_1`, `x`, and `v`. We choose to color `v` with 1.



We color the remaining two variables, `tmp_1` and `x`, with 1.



So we obtain the following coloring:

$$\{\texttt{tmp\_0} \mapsto 0, \texttt{tmp\_1} \mapsto 1, \texttt{z} \mapsto 1, \texttt{x} \mapsto 1, \texttt{y} \mapsto 2, \texttt{w} \mapsto 0, \texttt{v} \mapsto 1\}$$

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 in the file `priority_queue.py` of the support code.

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 locations.

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

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

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

$$\texttt{z} \mapsto \texttt{-8(\%rbp)},\ \texttt{tmp\_0} \mapsto \texttt{\%rcx},\ \texttt{tmp\_1} \mapsto \texttt{-8(\%rbp)}\}$$

Adapt the code from the `assign_homes` pass (Section 2.6) 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.

```
movq $1, v                  movq $1, -8(%rbp)
movq $42, w                 movq $42, %rcx
movq v, x                   movq -8(%rbp), -8(%rbp)
addq $7, x                  addq $7, -8(%rbp)
movq x, y                   movq -8(%rbp), -16(%rbp)
movq x, z                   movq -8(%rbp), -8(%rbp)
addq w, z            ⇒      addq %rcx, -8(%rbp)
movq y, tmp_0               movq -16(%rbp), %rcx
negq tmp_0                  negq %rcx
movq z, tmp_1               movq -8(%rbp), -8(%rbp)
addq tmp_0, tmp_1           addq %rcx, -8(%rbp)
movq tmp_1, %rdi            movq -8(%rbp), %rdi
callq print_int             callq print_int
```

Exercise 13 Implement the `allocate_registers` pass. Create five programs that exercise all aspects of the register allocation algorithm, including spilling variables to the stack. Run the `run-tests.py` script to to check whether the output programs produce the same result as the input programs.

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 2.7 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.

```
movq $1, -8(%rbp)
movq $42, %rcx                     movq $1, -8(%rbp)
movq -8(%rbp), -8(%rbp)            movq $42, %rcx
addq $7, -8(%rbp)                  addq $7, -8(%rbp)
movq -8(%rbp), -16(%rbp)           movq -8(%rbp), %rax
movq -8(%rbp), -8(%rbp)            movq %rax, -16(%rbp)
addq %rcx, -8(%rbp)         ⇒      addq %rcx, -8(%rbp)
movq -16(%rbp), %rcx               movq -16(%rbp), %rcx
negq %rcx                          negq %rcx
movq -8(%rbp), -8(%rbp)            addq %rcx, -8(%rbp)
addq %rcx, -8(%rbp)                movq -8(%rbp), %rdi
movq -8(%rbp), %rdi                callq print_int
callq print_int
```

Exercise 14 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 a field named `used_callee` to the `X86Program` AST node 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 16 bytes.

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

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`.

Figure 3.9 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`, `y`, and `tmp_0` were assigned to `rcx` and variables `w` and `tmp_1` were assigned to `rbx`. Variable `z` 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 z 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.

Exercise 15 Update the `prelude_and_conclusion` pass as described in this section. Run the script to test the complete compiler for $\mathcal{L}_{\mathsf{Var}}$ 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.

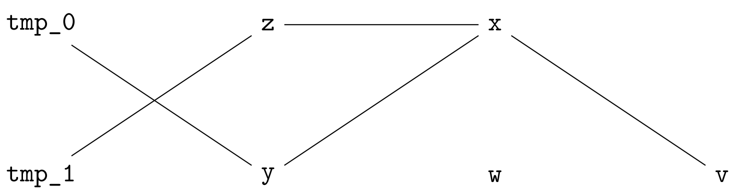
```
        .globl main
main:
        pushq %rbp
        movq %rsp, %rbp
        pushq %rbx
        subq $8, %rsp
        movq $1, %rcx
        movq $42, %rbx
        addq $7, %rcx
        movq %rcx, -16(%rbp)
        addq %rbx, -16(%rbp)
        negq %rcx
        movq -16(%rbp), %rbx
        addq %rcx, %rbx
        movq %rbx, %rdi
        callq print_int
        addq $8, %rsp
        popq %rbx
        popq %rbp
        retq
```

Figure 3.9
The x86 output from the running example (Figure 3.1), limiting allocation to just `rbx` and `rcx`.

To motivate the need for move biasing we return to the running example and recall that in Section 3.5 we were able to remove three trivial move instructions from the running example. However, we could remove another trivial move if we were able to allocate `y` and `tmp_0` to the same register.

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 before the coloring of `w`. Recall the following configuration. The most saturated vertices were `tmp_1`, `w`, and `y`.

$$\begin{tikzcd}
\mathtt{tmp_0}:0,\{1\} \arrow[r, no head] \arrow[d, no head] & \mathtt{z}:1,\{0\} \arrow[d, no head] \arrow[dr, no head] & \mathtt{x}:-,\{\} \arrow[d, no head] & \\
\mathtt{tmp_1}:-,\{0\} & \mathtt{y}:-,\{1\} \arrow[r, no head] & \mathtt{w}:-,\{1\} \arrow[r, no head] & \mathtt{v}:-,\{\}
\end{tikzcd}$$

We have arbitrarily chosen to color `w` instead of `tmp_1` or `y`, but note that `w` is not move related to any variables, whereas `y` and `tmp_1` are move related to `tmp_0` and `z`, respectively. If we instead choose `y` and color it 0, we can delete another move instruction.

$$\begin{tikzcd}
\mathtt{tmp_0}:0,\{1\} \arrow[r, no head] \arrow[d, no head] & \mathtt{z}:1,\{0\} \arrow[d, no head] \arrow[dr, no head] & \mathtt{x}:-,\{\} \arrow[d, no head] & \\
\mathtt{tmp_1}:-,\{0\} & \mathtt{y}:0,\{1\} \arrow[r, no head] & \mathtt{w}:-,\{0,1\} \arrow[r, no head] & \mathtt{v}:-,\{\}
\end{tikzcd}$$

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

$$\begin{tikzcd}
\mathtt{tmp_0}:0,\{1\} \arrow[r, no head] \arrow[d, no head] & \mathtt{z}:1,\{0\} \arrow[d, no head] \arrow[dr, no head] & \mathtt{x}:-,\{2\} \arrow[d, no head] & \\
\mathtt{tmp_1}:-,\{0\} & \mathtt{y}:0,\{1,2\} \arrow[r, no head] & \mathtt{w}:2,\{0,1\} \arrow[r, no head] & \mathtt{v}:-,\{2\}
\end{tikzcd}$$

To finish the coloring, `x` and `v` get 0 and `tmp_1` gets 1.

$$\begin{tikzcd}
\mathtt{tmp_0}:0,\{1\} \arrow[r, no head] \arrow[d, no head] & \mathtt{z}:1,\{0\} \arrow[d, no head] \arrow[dr, no head] & \mathtt{x}:0,\{2\} \arrow[d, no head] & \\
\mathtt{tmp_1}:1,\{0\} & \mathtt{y}:0,\{1,2\} \arrow[r, no head] & \mathtt{w}:2,\{0,1\} \arrow[r, no head] & \mathtt{v}:0,\{2\}
\end{tikzcd}$$

So we have the following assignment of variables to registers.

$$\{\mathtt{v} \mapsto \mathtt{\%rcx},\ \mathtt{w} \mapsto \mathtt{-16(\%rbp)},\ \mathtt{x} \mapsto \mathtt{\%rcx},\ \mathtt{y} \mapsto \mathtt{\%rcx},$$
$$\mathtt{z} \mapsto \mathtt{-8(\%rbp)},\ \mathtt{tmp\_0} \mapsto \mathtt{\%rcx},\ \mathtt{tmp\_1} \mapsto \mathtt{-8(\%rbp)}\}$$

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.

```
movq $1, v
movq $42, w
movq v, x
addq $7, x
movq x, y
movq x, z
addq w, z
movq y, tmp_0
negq tmp_0
movq z, tmp_1
addq tmp_0, tmp_1
movq tmp_1, %rdi
callq _print_int
```

⇒

```
movq $1, %rcx
movq $42, -16(%rbp)
movq %rcx, %rcx
addq $7, %rcx
movq %rcx, %rcx
movq %rcx, -8(%rbp)
addq -16(%rbp), -8(%rbp)
movq %rcx, %rcx
negq %rcx
movq -8(%rbp), -8(%rbp)
addq %rcx, -8(%rbp)
movq -8(%rbp), %rdi
callq _print_int
```

⇒

```
movq $1, %rcx
movq $42, -16(%rbp)
addq $7, %rcx
movq %rcx, -8(%rbp)
movq -16(%rbp), %rax
addq %rax, -8(%rbp)
negq %rcx
addq %rcx, -8(%rbp)
movq -8(%rbp), %rdi
callq print_int
```

Exercise 16 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 $\mathcal{L}_{\mathsf{Var}}$ 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 $\mathcal{L}_{\mathsf{If}}$ language. The Boolean values *true* and *false* are written `True` and `False` respectively in Python. The $\mathcal{L}_{\mathsf{If}}$ language includes several operations that involve Booleans (`and`, `not`, $==$, `<`, etc.) and the `if` expression and statement. With the addition of `if`, programs can have non-trivial control flow which impacts liveness analysis and motivates a new pass named `explicate_control`. 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. Python uses a mixture of these two options, depending on the operation and the kind of value. For example, the result of `not 1` is False because Python treats non-zero integers as if they were `True`. On the other hand, `1[0]` results in a run-time error in Python because an "`int` object is not subscriptable".

The MyPy type checker makes similar design choices as Python, except much of the error detection happens at compile time instead of run time (Lehtosalo 2021). MyPy accepts `not 1`. But in the case of `1[0]`, MyPy reports a compile-time error stating that a "value of type `int` is not indexable".

The $\mathcal{L}_{\mathsf{If}}$ language performs type checking during compilation like MyPy. In Chapter 9 we study the alternative choice, that is, a dynamically typed language like Python. The $\mathcal{L}_{\mathsf{If}}$ language is a subset of MyPy; for some operations we are more restrictive, for example, rejecting `not 1`. We keep the type checker for $\mathcal{L}_{\mathsf{If}}$ 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 $\mathcal{L}_{\mathsf{If}}$ language (Section 4.1). We then introduce the idea of type checking and define a type checker for $\mathcal{L}_{\mathsf{If}}$ (Section 4.2). 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 translates some operators into others, thereby reducing the number of operators that need to be handled in later passes. The main event of this chapter is the `explicate_control` pass that is responsible for translating `if`'s into conditional

$$
\begin{array}{rcl}
exp & ::= & int \mid \texttt{input\_int()} \mid \texttt{-}exp \mid exp \texttt{+} exp \mid exp \texttt{-} exp \mid \texttt{(}exp\texttt{)} \\
stmt & ::= & \texttt{print(}exp\texttt{)} \mid exp \\
\hline
exp & ::= & var \\
stmt & ::= & var \texttt{=} exp \\
\hline
cmp & ::= & \texttt{==} \mid \texttt{!=} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \\
exp & ::= & \texttt{True} \mid \texttt{False} \mid exp \texttt{ and } exp \mid exp \texttt{ or } exp \mid \texttt{not } exp \\
 & \mid & exp\ cmp\ exp \mid exp \texttt{ if } exp \texttt{ else } exp \\
stmt & ::= & \texttt{if } exp\texttt{: } stmt^{+} \texttt{ else: } stmt^{+} \\
\mathcal{L}_{\mathsf{If}} & ::= & stmt^{*}
\end{array}
$$

Figure 4.1
The concrete syntax of $\mathcal{L}_{\mathsf{If}}$, extending $\mathcal{L}_{\mathsf{Var}}$ (Figure 2.1) with Booleans and conditionals.

`goto`'s (Section 4.7). Regarding register allocation, there is the interesting question of how to handle conditional `goto`'s during liveness analysis.

4.1 The $\mathcal{L}_{\mathsf{If}}$ Language

The concrete and abstract syntax of the $\mathcal{L}_{\mathsf{If}}$ language are defined in Figures 4.1 and 4.2, respectively. The $\mathcal{L}_{\mathsf{If}}$ language includes all of $\mathcal{L}_{\mathsf{Var}}$ (shown in gray), the Boolean literals `True` and `False`, the `if` expression, and the `if` statement. We expand the set of operators to include

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

Figure 4.3 defines the interpreter for $\mathcal{L}_{\mathsf{If}}$, which inherits from the interpreter for $\mathcal{L}_{\mathsf{Var}}$ (Figure 2.4). The literals `True` and `False` evaluate to the corresponding Boolean values. The conditional expression e_2 `if` e_1 `else` e_3 evaluates expression e_1 and then either evaluates e_2 or e_3 depending on whether e_1 produced `True` or `False`. 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 e_1 `and` e_2, the expression e_2 is not evaluated if e_1 evaluates to `False`. Similarly, given e_1 `or` e_2, the expression e_2 is not evaluated if e_1 evaluates to `True`.

4.2 Type Checking $\mathcal{L}_{\mathsf{If}}$ 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 $\mathcal{L}_{\mathsf{If}}$, we have just two types, `int` and `bool`. So a type checker should predict that

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

$$
\begin{array}{rcl}
\mathit{binaryop} & ::= & \texttt{Add()} \mid \texttt{Sub()} \\
\mathit{unaryop} & ::= & \texttt{USub()} \\
\mathit{exp} & ::= & \texttt{Constant(}\mathit{int}\texttt{)} \mid \texttt{Call(Name('input\_int'),[])} \\
 & \mid & \texttt{UnaryOp(}\mathit{unaryop}\texttt{,}\mathit{exp}\texttt{)} \mid \texttt{BinOp(}\mathit{binaryop}\texttt{,}\mathit{exp}\texttt{,}\mathit{exp}\texttt{)} \\
\mathit{stmt} & ::= & \texttt{Expr(Call(Name('print'),[}\mathit{exp}\texttt{]))} \mid \texttt{Expr(}\mathit{exp}\texttt{)} \\
\hline
\mathit{exp} & ::= & \texttt{Name(}\mathit{var}\texttt{)} \\
\mathit{stmt} & ::= & \texttt{Assign([Name(}\mathit{var}\texttt{)], }\mathit{exp}\texttt{)} \\
\hline
\mathit{boolop} & ::= & \texttt{And()} \mid \texttt{Or()} \\
\mathit{unaryop} & ::= & \texttt{Not()} \\
\mathit{cmp} & ::= & \texttt{Eq()} \mid \texttt{NotEq()} \mid \texttt{Lt()} \mid \texttt{LtE()} \mid \texttt{Gt()} \mid \texttt{GtE()} \\
\mathit{bool} & ::= & \texttt{True} \mid \texttt{False} \\
\mathit{exp} & ::= & \texttt{Constant(}\mathit{bool}\texttt{)} \mid \texttt{BoolOp(}\mathit{boolop}\texttt{,[}\mathit{exp}\texttt{,}\mathit{exp}\texttt{])} \\
 & \mid & \texttt{Compare(}\mathit{exp}\texttt{,[}\mathit{cmp}\texttt{],[}\mathit{exp}\texttt{])} \mid \texttt{IfExp(}\mathit{exp}\texttt{,}\mathit{exp}\texttt{,}\mathit{exp}\texttt{)} \\
\mathit{stmt} & ::= & \texttt{If(}\mathit{exp}\texttt{, }\mathit{stmt}^{+}\texttt{, }\mathit{stmt}^{+}\texttt{)} \\
\mathcal{L}_{\mathsf{If}} & ::= & \texttt{Module(}\mathit{stmt}^{*}\texttt{)}
\end{array}
$$

Figure 4.2
The abstract syntax of $\mathcal{L}_{\mathsf{If}}$.

produces a value of type `int` while

```
(not False) and True
```

produces a value of type `bool`.

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 $\mathcal{L}_{\mathsf{If}}$ signals an error for the below expression

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

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

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 $\mathcal{L}_{\mathsf{Var}}$ subset into its own class, shown in Figure 4.5. The type checker for $\mathcal{L}_{\mathsf{If}}$ is shown in Figure 4.6 and it inherits from the type checker for $\mathcal{L}_{\mathsf{Var}}$. These type checkers are in the files `type_check_Lvar.py` and `type_check_Lif.py` of the support code. Each type checker is a structurally recursive function over the AST. Given an input expression `e`, the type checker either signals an error or returns its type.

Next we discuss the `type_check_exp` function of $\mathcal{L}_{\mathsf{Var}}$ in Figure 4.5. The type of an integer constant is `int`. To handle variables, the type checker uses the environment `env` to map variables to types. Consider the case for assignment. We type check the initializing expression to obtain its type `t`. If the variable `lhs.id`

```
class InterpLif(InterpLvar):

  def interp_exp(self, e, env):
    match e:
      case IfExp(test, body, orelse):
        if self.interp_exp(test, env):
          return self.interp_exp(body, env)
        else:
          return self.interp_exp(orelse, env)
      case UnaryOp(Not(), v):
        return not self.interp_exp(v, env)
      case BoolOp(And(), values):
        if self.interp_exp(values[0], env):
          return self.interp_exp(values[1], env)
        else:
          return False
      case BoolOp(Or(), values):
        if self.interp_exp(values[0], env):
          return True
        else:
          return self.interp_exp(values[1], env)
      case Compare(left, [cmp], [right]):
        l = self.interp_exp(left, env)
        r = self.interp_exp(right, env)
        return self.interp_cmp(cmp)(l, r)
      case _:
        return super().interp_exp(e, env)

  def interp_stmts(self, ss, env):
    if len(ss) == 0:
      return
    match ss[0]:
      case If(test, body, orelse):
        if self.interp_exp(test, env):
          return self.interp_stmts(body + ss[1:], env)
        else:
          return self.interp_stmts(orelse + ss[1:], env)
      case _:
        return super().interp_stmts(ss, env)
  ...
```

Figure 4.3
Interpreter for the $\mathcal{L}_{\mathsf{If}}$ language. (See Figure 4.4 for `interp_cmp`.)

is already in the environment because there was a prior assignment, we check that this initializer has the same type as the prior one. If this is the first assignment to the variable, we associate type `t` with the variable `lhs.id` in the environment. Thus, when the type checker encounters a use of variable `x`, it can find its type in the environment. Regarding addition, subtraction, and negation, we recursively analyze the arguments, check that they have type `int`, and return `int`.

```
class InterpLif(InterpLvar):
  ...
  def interp_cmp(self, cmp):
    match cmp:
      case Lt():
        return lambda x, y: x < y
      case LtE():
        return lambda x, y: x <= y
      case Gt():
        return lambda x, y: x > y
      case GtE():
        return lambda x, y: x >= y
      case Eq():
        return lambda x, y: x == y
      case NotEq():
        return lambda x, y: x != y
```

Figure 4.4
Interpreter for the comparison operators in the $\mathcal{L}_{\mathsf{If}}$ language.

The auxiliary method `check_type_equal` triggers an error if the two types are not equal.

The type checker for $\mathcal{L}_{\mathsf{If}}$ is defined in Figure 4.6. The type of a Boolean constant is `bool`. Logical not requires its argument to be a `bool` and produces a `bool`. Similarly for logical and and logical or. The equality operator requires the two arguments to have the same type and therefore we handle it separately from the other operators. The other comparisons (less-than, etc.) require their arguments to be of type `int` and they produce a `bool`. The condition of an `if` must be of `bool` type and the two branches must have the same type.

Exercise 17 Create 10 new test programs in $\mathcal{L}_{\mathsf{If}}$. 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.py` is expected to error, then create an empty file named `cond_test_14.tyerr`. The other half of the test programs should not have type errors. Run the test script to check that these test programs type check as expected.

```
class TypeCheckLvar:
  def check_type_equal(self, t1, t2, e):
    if t1 != t2:
      msg = 'error: ' + repr(t1) + ' != ' + repr(t2) + ' in ' + repr(e)
      raise Exception(msg)

  def type_check_exp(self, e, env):
    match e:
      case BinOp(left, (Add() | Sub()), right):
        l = self.type_check_exp(left, env)
        check_type_equal(l, int, left)
        r = self.type_check_exp(right, env)
        check_type_equal(r, int, right)
        return int
      case UnaryOp(USub(), v):
        t = self.type_check_exp(v, env)
        check_type_equal(t, int, v)
        return int
      case Name(id):
        return env[id]
      case Constant(value) if isinstance(value, int):
        return int
      case Call(Name('input_int'), []):
        return int

  def type_check_stmts(self, ss, env):
    if len(ss) == 0:
      return
    match ss[0]:
      case Assign([lhs], value):
        t = self.type_check_exp(value, env)
        if lhs.id in env:
          check_type_equal(env[lhs.id], t, value)
        else:
          env[lhs.id] = t
        return self.type_check_stmts(ss[1:], env)
      case Expr(Call(Name('print'), [arg])):
        t = self.type_check_exp(arg, env)
        check_type_equal(t, int, arg)
        return self.type_check_stmts(ss[1:], env)
      case Expr(value):
        self.type_check_exp(value, env)
        return self.type_check_stmts(ss[1:], env)

  def type_check_P(self, p):
    match p:
      case Module(body):
        self.type_check_stmts(body, {})
```

Figure 4.5
Type checker for the $\mathcal{L}_{\mathsf{Var}}$ language.

```
class TypeCheckLif(TypeCheckLvar):
  def type_check_exp(self, e, env):
    match e:
      case Constant(value) if isinstance(value, bool):
        return bool
      case BinOp(left, Sub(), right):
        l = self.type_check_exp(left, env); check_type_equal(l, int, left)
        r = self.type_check_exp(right, env); check_type_equal(r, int, right)
        return int
      case UnaryOp(Not(), v):
        t = self.type_check_exp(v, env); check_type_equal(t, bool, v)
        return bool
      case BoolOp(op, values):
        left = values[0] ; right = values[1]
        l = self.type_check_exp(left, env); check_type_equal(l, bool, left)
        r = self.type_check_exp(right, env); check_type_equal(r, bool, right)
        return bool
      case Compare(left, [cmp], [right]) if isinstance(cmp, Eq) \
                                        or isinstance(cmp, NotEq):
        l = self.type_check_exp(left, env)
        r = self.type_check_exp(right, env)
        check_type_equal(l, r, e)
        return bool
      case Compare(left, [cmp], [right]):
        l = self.type_check_exp(left, env); check_type_equal(l, int, left)
        r = self.type_check_exp(right, env); check_type_equal(r, int, right)
        return bool
      case IfExp(test, body, orelse):
        t = self.type_check_exp(test, env); check_type_equal(bool, t, test)
        b = self.type_check_exp(body, env)
        o = self.type_check_exp(orelse, env)
        check_type_equal(b, o, e)
        return b
      case _:
        return super().type_check_exp(e, env)

  def type_check_stmts(self, ss, env):
    if len(ss) == 0:
      return
    match ss[0]:
      case If(test, body, orelse):
        t = self.type_check_exp(test, env); check_type_equal(bool, t, test)
        b = self.type_check_stmts(body, env)
        o = self.type_check_stmts(orelse, env)
        check_type_equal(b, o, ss[0])
        return self.type_check_stmts(ss[1:], env)
      case _:
        return super().type_check_stmts(ss, env)
```

Figure 4.6
Type checker for the $\mathcal{L}_{\mathsf{If}}$ language.

$$
\begin{array}{lcl}
\mathit{atm} & ::= & \mathit{int} \mid \mathit{var} \mid \mathit{bool} \\
\mathit{exp} & ::= & \mathit{atm} \mid \texttt{input\_int()} \mid \mathit{atm}\ \mathit{binaryop}\ \mathit{atm} \mid \mathit{unaryop}\ \mathit{atm} \\
 & \mid & \mathit{atm}\ \mathit{cmp}\ \mathit{atm} \\
\mathit{stmt} & ::= & \texttt{print(}\mathit{atm}\texttt{)} \mid \mathit{exp} \\
 & \mid & \mathit{var}\ \texttt{=}\ \mathit{exp} \mid \texttt{return}\ \mathit{exp} \mid \texttt{goto}\ \mathit{label} \\
 & \mid & \texttt{if}\ \mathit{atm}\ \mathit{cmp}\ \mathit{atm}\texttt{:}\ \texttt{goto}\ \mathit{label}\ \texttt{else:}\ \texttt{goto}\ \mathit{label} \\
\mathcal{C}_{\mathsf{If}} & ::= & (\mathit{label}\texttt{:}\ \mathit{stmt}^*)\ldots
\end{array}
$$

Figure 4.7
The concrete syntax of the $\mathcal{C}_{\mathsf{If}}$ intermediate language, an extension of $\mathcal{C}_{\mathsf{Var}}$ (Figure **??**).

$$
\begin{array}{lcl}
\mathit{atm} & ::= & \texttt{Constant(}\mathit{int}\texttt{)} \mid \texttt{Name(}\mathit{var}\texttt{)} \mid \texttt{Constant(}\mathit{bool}\texttt{)} \\
\mathit{exp} & ::= & \mathit{atm} \mid \texttt{Call(Name('input\_int'),[])} \\
 & \mid & \texttt{BinOp(}\mathit{atm}\texttt{,}\mathit{binaryop}\texttt{,}\mathit{atm}\texttt{)} \mid \texttt{UnaryOp(}\mathit{unaryop}\texttt{,}\mathit{atm}\texttt{)} \\
 & \mid & \texttt{Compare(}\mathit{atm}\texttt{,[}\mathit{cmp}\texttt{],[}\mathit{atm}\texttt{])} \\
\mathit{stmt} & ::= & \texttt{Expr(Call(Name('print'),[}\mathit{atm}\texttt{]))} \mid \texttt{Expr(}\mathit{exp}\texttt{)} \\
 & \mid & \texttt{Assign([Name(}\mathit{var}\texttt{)], }\mathit{exp}\texttt{)} \mid \texttt{Return(}\mathit{exp}\texttt{)} \mid \texttt{Goto(}\mathit{label}\texttt{)} \\
 & \mid & \texttt{If(Compare(}\mathit{atm}\texttt{,[}\mathit{cmp}\texttt{],[}\mathit{atm}\texttt{]), [Goto(}\mathit{label}\texttt{)], [Goto(}\mathit{label}\texttt{)])} \\
\mathcal{C}_{\mathsf{If}} & ::= & \texttt{CProgram(\{}\mathit{label}\texttt{:}\ \mathit{stmt}^*\texttt{, }\ldots\texttt{\})}
\end{array}
$$

Figure 4.8
The abstract syntax of $\mathcal{C}_{\mathsf{If}}$.

4.3 The $\mathcal{C}_{\mathsf{If}}$ Intermediate Language

The output of `explicate_control` is a language similar to the *C* language (Kernighan and Ritchie 1988) in that it has labels and `goto` statements, so we name it $\mathcal{C}_{\mathsf{If}}$. The $\mathcal{C}_{\mathsf{If}}$ language supports the same operators as $\mathcal{L}_{\mathsf{If}}$ but the arguments of operators are restricted to atomic expressions. The $\mathcal{C}_{\mathsf{If}}$ language does not include `if` expressions but it does include a restricted form of `if` statment. The condition must be a comparison and the two branches may only contain `goto` statements. These restrictions make it easier to translate `if` statements to x86. The $\mathcal{C}_{\mathsf{If}}$ language also adds a `return` statement to finish the program with a specified value. The `CProgram` construct contains a dictionary mapping labels to lists of statements that end with a `return` statement, a `goto`, or a conditional `goto`. A `goto` statement transfers control to the sequence of statements associated with its label. The concrete syntax for $\mathcal{C}_{\mathsf{If}}$ is defined in Figure 4.7 and the abstract syntax is defined in Figure 4.8.

4.4 The $\mathrm{x86}_{\mathsf{If}}$ Language

To implement the new logical operations, the comparison operations, and the `if` expression and statement, we delve further into the x86 language. Figures 4.9 and 4.10 define the concrete and abstract syntax for the $\mathrm{x86}_{\mathsf{If}}$ subset of x86, which

$$
\begin{array}{lcl}
\mathit{reg} & ::= & \texttt{rsp} \mid \texttt{rbp} \mid \texttt{rax} \mid \texttt{rbx} \mid \texttt{rcx} \mid \texttt{rdx} \mid \texttt{rsi} \mid \texttt{rdi} \mid \\
 & & \texttt{r8} \mid \texttt{r9} \mid \texttt{r10} \mid \texttt{r11} \mid \texttt{r12} \mid \texttt{r13} \mid \texttt{r14} \mid \texttt{r15} \\
\mathit{arg} & ::= & \texttt{\$}\mathit{int} \mid \texttt{\%}\mathit{reg} \mid \mathit{int}\texttt{(}\texttt{\%}\mathit{reg}\texttt{)} \\
\mathit{instr} & ::= & \texttt{addq}\ \mathit{arg}\texttt{,}\mathit{arg} \mid \texttt{subq}\ \mathit{arg}\texttt{,}\mathit{arg} \mid \texttt{negq}\ \mathit{arg} \mid \texttt{movq}\ \mathit{arg}\texttt{,}\mathit{arg} \mid \\
 & & \texttt{pushq}\ \mathit{arg} \mid \texttt{popq}\ \mathit{arg} \mid \texttt{callq}\ \mathit{label} \mid \texttt{retq} \mid \texttt{jmp}\ \mathit{label} \mid \\
 & & \mathit{label}\texttt{:}\ \mathit{instr} \\
\hline
\mathit{bytereg} & ::= & \texttt{ah} \mid \texttt{al} \mid \texttt{bh} \mid \texttt{bl} \mid \texttt{ch} \mid \texttt{cl} \mid \texttt{dh} \mid \texttt{dl} \\
\mathit{arg} & ::= & \texttt{\%}\mathit{bytereg} \\
\mathit{cc} & ::= & \texttt{e} \mid \texttt{ne} \mid \texttt{l} \mid \texttt{le} \mid \texttt{g} \mid \texttt{ge} \\
\mathit{instr} & ::= & \texttt{xorq}\ \mathit{arg}\texttt{,}\ \mathit{arg} \mid \texttt{cmpq}\ \mathit{arg}\texttt{,}\ \mathit{arg} \mid \texttt{set}\mathit{cc}\ \mathit{arg} \mid \texttt{movzbq}\ \mathit{arg}\texttt{,}\ \mathit{arg} \\
 & \mid & \texttt{j}\mathit{cc}\ \mathit{label} \\
\mathrm{x86}_{\mathsf{If}} & ::= & \texttt{.globl main} \\
 & & \texttt{main:}\ \mathit{instr} \ldots
\end{array}
$$

Figure 4.9
The concrete syntax of $\mathrm{x86}_{\mathsf{If}}$ (extends $\mathrm{x86}_{\mathsf{Int}}$ of Figure 2.5).

includes instructions for logical operations, comparisons, and jumps. The abstract syntax for an $\mathrm{x86}_{\mathsf{If}}$ program contains a dictionary mapping labels to sequences of instructions, each of which we refer to as a *basic block*.

One challenge is that x86 does not provide an instruction that directly implements logical negation (`not` in $\mathcal{L}_{\mathsf{If}}$ and $\mathcal{C}_{\mathsf{If}}$). 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.

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

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

$$
\begin{array}{lcl}
\mathit{bytereg} & ::= & \texttt{'ah'} \mid \texttt{'al'} \mid \texttt{'bh'} \mid \texttt{'bl'} \mid \texttt{'ch'} \mid \texttt{'cl'} \mid \texttt{'dh'} \mid \texttt{'dl'} \\
\mathit{arg} & ::= & \texttt{Immediate(}\mathit{int}\texttt{)} \mid \texttt{Reg(}\mathit{reg}\texttt{)} \mid \texttt{Deref(}\mathit{reg}\texttt{,}\mathit{int}\texttt{)} \mid \texttt{ByteReg(}\mathit{bytereg}\texttt{)} \\
\mathit{cc} & ::= & \texttt{'e'} \mid \texttt{'ne'} \mid \texttt{'l'} \mid \texttt{'le'} \mid \texttt{'g'} \mid \texttt{'ge'} \\
\mathit{instr} & ::= & \texttt{Instr('addq',[}\mathit{arg}\texttt{,}\mathit{arg}\texttt{])} \mid \texttt{Instr('subq',[}\mathit{arg}\texttt{,}\mathit{arg}\texttt{])} \\
& \mid & \texttt{Instr('movq',[}\mathit{arg}\texttt{,}\mathit{arg}\texttt{])} \mid \texttt{Instr('negq',[}\mathit{arg}\texttt{])} \\
& \mid & \texttt{Callq(}\mathit{label}\texttt{,}\mathit{int}\texttt{)} \mid \texttt{Retq()} \mid \texttt{Instr('pushq',[}\mathit{arg}\texttt{])} \\
& \mid & \texttt{Instr('popq',[}\mathit{arg}\texttt{])} \mid \texttt{Jump(}\mathit{label}\texttt{)} \\
& \mid & \texttt{Instr('xorq',[}\mathit{arg}\texttt{,}\mathit{arg}\texttt{])} \mid \texttt{Instr('cmpq',[}\mathit{arg}\texttt{,}\mathit{arg}\texttt{])} \\
& \mid & \texttt{Instr('set',[}\mathit{cc}\texttt{,}\mathit{arg}\texttt{])} \mid \texttt{Instr('movzbq',[}\mathit{arg}\texttt{,}\mathit{arg}\texttt{])} \\
& \mid & \texttt{JumpIf(}\mathit{cc}\texttt{,}\mathit{label}\texttt{)} \\
\mathit{block} & ::= & \mathit{instr}^{+} \\
\mathrm{x86}_{\mathsf{If}} & ::= & \texttt{X86Program(\{}\mathit{label} : \mathit{block}\texttt{,} \ldots \texttt{\})}
\end{array}
$$

Figure 4.10
The abstract syntax of $\mathrm{x86}_{\mathsf{If}}$ (extends $\mathrm{x86}_{\mathsf{Int}}$ of Figure 2.9).

register. This register cannot be accessed directly but it can be queried by a number of instructions, including the `set` instruction. The instruction `set`*cc 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, `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 instructions for jumping are relevant to the compilation of `if` expressions. The instruction `jmp` *label* updates the program counter to the address of the instruction after the specified label. The instruction `j`*cc 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, `JumpIf('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 $\mathcal{L}_{\mathsf{If}}$ Language

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

$$e_1 \ \texttt{and} \ e_2 \quad \Rightarrow \quad e_2 \ \texttt{if} \ e_1 \ \texttt{else False}$$

$$e_1 \ \texttt{or} \ e_2 \quad \Rightarrow \quad \texttt{True if} \ e_1 \ \texttt{else} \ e_2$$

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

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

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

Exercise 18 Implement the pass `shrink` to remove `and` and `or` from the language by translating them to `if` expressions in $\mathcal{L}_{\mathsf{If}}$. Create four test programs that involve these operators. Run the script to test your compiler on all the test programs.

4.6 Remove Complex Operands

The output language of `remove_complex_operands` is $\mathcal{L}^{mon}_{\mathsf{If}}$ (Figure 4.11), the monadic normal form of $\mathcal{L}_{\mathsf{If}}$. 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. We add a new language form, the `Begin` expression, to aid in the translation of `if` expressions. When we recursively process the two branches of the `if`, we generate temporary variables and their initializing expressions. However, these expressions may contain side effects and should only be executed when the condition of the `if` is true (for the "then" branch) or false (for the "else" branch). The `Begin` provides a way to initialize the temporary variables within the two branches of the `if` expression. In general, the `Begin(`*ss*`, `*e*`)` form execute the statements *ss* and then returns the result of expression *e*.

Add cases to the `rco_exp` and `rco_atom` functions for the new features in $\mathcal{L}_{\mathsf{If}}$. When recursively processing subexpressions, recall that you should invoke `rco_atom` when the output needs to be an *atm* (as specified in the grammar for $\mathcal{L}^{mon}_{\mathsf{If}}$) 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 19 Add cases for Boolean constants and `if` to the `rco_atom` and `rco_exp` functions in `compiler.rkt`. Create three new $\mathcal{L}_{\mathsf{If}}$ programs that exercise the interesting code in this pass.

4.7 Explicate Control

The `explicate_control` pass translates from $\mathcal{L}_{\mathsf{If}}$ to $\mathcal{C}_{\mathsf{If}}$. The main challenge to overcome is that the condition of an `if` can be an arbitrary expression in $\mathcal{L}_{\mathsf{If}}$ whereas in $\mathcal{C}_{\mathsf{If}}$ the condition must be a comparison.

$$
\begin{array}{rcl}
\mathit{atm} & ::= & \texttt{Constant(}\mathit{int}\texttt{)} \mid \texttt{Name(}\mathit{var}\texttt{)} \\
\mathit{exp} & ::= & \mathit{atm} \mid \texttt{Call(Name('input\_int'),[])} \\
& \mid & \texttt{UnaryOp(}\mathit{unaryop}\texttt{,}\mathit{atm}\texttt{)} \mid \texttt{BinOp(}\mathit{atm}\texttt{,}\mathit{binaryop}\texttt{,}\mathit{atm}\texttt{)} \\
\mathit{stmt} & ::= & \texttt{Expr(Call(Name('print'),[}\mathit{atm}\texttt{]))} \mid \texttt{Expr(}\mathit{exp}\texttt{)} \\
& \mid & \texttt{Assign([Name(}\mathit{var}\texttt{)], }\mathit{exp}\texttt{)} \\
\hline
\mathit{atm} & ::= & \texttt{Constant(}\mathit{bool}\texttt{)} \\
\mathit{exp} & ::= & \texttt{Compare(}\mathit{atm}\texttt{,[}\mathit{cmp}\texttt{],[}\mathit{atm}\texttt{])} \mid \texttt{IfExp(}\mathit{exp}\texttt{,}\mathit{exp}\texttt{,}\mathit{exp}\texttt{)} \\
& \mid & \texttt{Begin(}\mathit{stmt}^{*}\texttt{, }\mathit{exp}\texttt{)} \\
\mathit{stmt} & ::= & \texttt{If(}\mathit{exp}\texttt{, }\mathit{stmt}^{*}\texttt{, }\mathit{stmt}^{*}\texttt{)} \\
\mathcal{L}_{\mathsf{if}}^{mon} & ::= & \texttt{Module(}\mathit{stmt}^{*}\texttt{)}
\end{array}
$$

Figure 4.11
$\mathcal{L}_{\mathsf{if}}^{mon}$ is $\mathcal{L}_{\mathsf{If}}$ in monadic normal form (extends $\mathcal{L}_{\mathsf{Var}}^{mon}$in Figure 2.11).

As a motivating example, consider the following program that has an `if` expression nested in the condition of another `if`.[4]

```
x = input_int()
y = input_int()
print(y + 2 if (x == 0 if x < 1 else x == 2) else 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`.

4. Programmers rarely write nested `if` expressions, but it is not uncommon for the condition of an `if` statement to be a call of a function that also contains an `if` statement. When such a function is inlined, the result is a nested `if` that requires the techniques discussed in this section.

```
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 $\mathcal{L}_{\mathsf{If}}$, pushing the outer `if` inside the inner one, yielding the following code.

```
x = input_int()
y = intput_int()
print(((y + 2) if x == 0 else (y + 10)) \
      if (x < 1) \
      else ((y + 2) if (x == 2) else (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 $\mathcal{C}_{\mathsf{If}}$ 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.

We recommend implementing `explicate_control` using the following four auxiliary functions.

`explicate_effect` generates code for expressions as statements, so their result is ignored and only their side effects matter.

`explicate_assign` generates code for expressions on the right-hand side of an assignment.

`explicate_pred` generates code for an `if` expression or statement by analyzing the condition expression.

`explicate_stmt` generates code for statements.

These four functions should build the dictionary of basic blocks. The following auxiliary function can be used to create a new basic block from a list of statements. It returns a `goto` statement that jumps to the new basic block.

```
x = input_int()
y = input_int()
print(y + 2           \
      if (x == 0      \
          if x < 1    \
          else x == 2) \
      else y + 10)
```

⇒

```
start:
  x = input_int()
  y = input_int()
  if x < 1:
    goto block_8
  else:
    goto block_9
block_8:
  if x == 0:
    goto block_4
  else:
    goto block_5
block_9:
  if x == 2:
    goto block_6
  else:
    goto block_7
block_4:
  goto block_2
block_5:
  goto block_3
block_6:
  goto block_2
block_7:
  goto block_3
block_2:
  tmp_0 = y + 2
  goto block_1
block_3:
  tmp_0 = y + 10
  goto block_1
block_1:
  print(tmp_0)
  return 0
```

Figure 4.12
Translation from $\mathcal{L}_{\mathsf{If}}$ to $\mathcal{C}_{\mathsf{If}}$ via the `explicate_control`.

```
def create_block(stmts, basic_blocks):
    label = label_name(generate_name('block'))
    basic_blocks[label] = stmts
    return Goto(label)
```

Figure 4.13 provides a skeleton for the `explicate_control` pass.

The `explicate_effect` function has three parameters: 1) the expression to be compiled, 2) the already-compiled code for this expression's *continuation*, that is, the list of statements that should execute after this expression, and 3) the dictionary of generated basic blocks. The `explicate_effect` function returns a list of $\mathcal{C}_{\mathsf{If}}$ statements and it may add to the dictionary of basic blocks. Let's consider a few of the cases for the expression to be compiled. If the expression to be compiled is a constant, then it can be discarded because it has no side effects. If it's a `input_int()`, then it has a side-effect and should be preserved. So the expression should be translated into a statement using the `Expr` AST class. If the expression to be compiled is an `if` expression, we translate the two branches using `explicate_effect` and then translate the condition expression using `explicate_pred`, which generates code for the entire `if`.

The `explicate_assign` function has four parameters: 1) the right-hand-side of the assignment, 2) the left-hand-side of the assignment (the variable), 3) the continuation, and 4) the dictionary of basic blocks. The `explicate_assign` function returns a list of $\mathcal{C}_{\mathsf{If}}$ statements and it may add to the dictionary of basic blocks.

When the right-hand-side is an `if` expression, there is some work to do. In particular, the two branches should be translated using `explicate_assign` and the condition expression should be translated using `explicate_pred`. Otherwise we can simply generate an assignment statement, with the given left and right-hand sides, concatenated with its continuation.

The `explicate_pred` function has four parameters: 1) the condition expression, 2) the generated statements for the "then" branch, 3) the generated statements for the "else" branch, and 4) the dictionary of basic blocks. The `explicate_pred` function returns a list of $\mathcal{C}_{\mathsf{If}}$ statements and it may add to the dictionary of basic blocks.

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 `x == 0`.

```
x = input_int()
42 if x == 0 else 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 `x == 0` to the following `if` statement.

```
if x == 0:
   goto block_1;
else
   goto block_2;
```

```
def explicate_effect(e, cont, basic_blocks):
    match e:
        case IfExp(test, body, orelse):
            ...
        case Call(func, args):
            ...
        case Begin(body, result):
            ...
        case _:
            ...

def explicate_assign(rhs, lhs, cont, basic_blocks):
    match rhs:
        case IfExp(test, body, orelse):
            ...
        case Begin(body, result):
            ...
        case _:
            return [Assign([lhs], rhs)] + cont

def explicate_pred(cnd, thn, els, basic_blocks):
    match cnd:
        case Compare(left, [op], [right]):
            goto_thn = create_block(thn, basic_blocks)
            goto_els = create_block(els, basic_blocks)
            return [If(cnd, [goto_thn], [goto_els])]
        case Constant(True):
            return thn;
        case Constant(False):
            return els;
        case UnaryOp(Not(), operand):
            ...
        case IfExp(test, body, orelse):
            ...
        case Begin(body, result):
            ...
        case _:
            return [If(Compare(cnd, [Eq()], [Constant(False)]),
                       [create_block(els, basic_blocks)],
                       [create_block(thn, basic_blocks)])]

def explicate_stmt(s, cont, basic_blocks):
    match s:
        case Assign([lhs], rhs):
            return explicate_assign(rhs, lhs, cont, basic_blocks)
        case Expr(value):
            return explicate_effect(value, cont, basic_blocks)
        case If(test, body, orelse):
            ...

def explicate_control(p):
    match p:
        case Module(body):
            new_body = [Return(Constant(0))]
            basic_blocks = {}
            for s in reversed(body):
                new_body = explicate_stmt(s, new_body, basic_blocks)
            basic_blocks[label_name('start')] = new_body
            return CProgram(basic_blocks)
```

Figure 4.13
Skeleton for the `explicate_control` pass.

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 True or False. Let us illustrate this with the following program.

```
42 if True else 777
```

Again, the two branches 42 and 777 were compiled to return statements, so explicate_pred compiles the constant True 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 body and orelse branches of the if inherit their context from the current one, that is, predicate context. So you should recursively apply explicate_pred to the body and orelse 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.

The last of the auxiliary functions is explicate_stmt. It has three parameters: 1) the statement to be compiled, 2) the code for its continuation, and 3) the dictionary of basic blocks. The explicate_stmt returns a list of statements and it may add to the dictionary of basic blocks. The cases for assignment and an expression-statement are given in full in the skeleton code: they simply dispatch to explicate_assign and explicate_effect, respectively. The case for if statements is not given, and is similar to the case for if expressions.

The explicate_control function itself is given in Figure 4.13. It applies explicate_stmt to each statement in the program, from back to front. Thus, the result so-far, stored in new_body, can be used as the continuation parameter in the next call to explicate_stmt. The new_body is initialized to a Return statement. Once complete, we add the new_body to the dictionary of basic blocks, labeling it as the "start" block.

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 input_int() 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 x == 0 and x == 2, respectively. In particular, we start block_4 with the comparison 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. The block_1 corresponds to the print statment at the end of the program.

Exercise 20 Implement `explicate_control` pass with its four auxiliary functions. Create test cases that exercise all of the new cases in the code for this pass.

4.8 Select Instructions

The `select_instructions` pass translates $\mathcal{C}_{\mathsf{If}}$ to $\mathrm{x86}^{\mathsf{Var}}_{\mathsf{If}}$. We begin with the Boolean constants. We take the usual approach of encoding them as integers.

$$\texttt{True} \quad\Rightarrow\quad \texttt{1} \qquad\qquad \texttt{False} \quad\Rightarrow\quad \texttt{0}$$

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.

$$\mathit{var}\ \texttt{= not}\ \mathit{var} \quad\Rightarrow\quad \texttt{xorq \$1,}\ \mathit{var}$$

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.

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

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.

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

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

A `goto` statement becomes a jump instruction.

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

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).

$$\begin{array}{l}\texttt{if}\ \mathit{atm}_1\ \texttt{==}\ \mathit{atm}_2\texttt{:}\\ \quad\texttt{goto}\ \ell_1\\ \texttt{else:}\\ \quad\texttt{goto}\ \ell_2\end{array} \quad\Rightarrow\quad \begin{array}{l}\texttt{cmpq}\ \mathit{arg}_2\texttt{,}\ \mathit{arg}_1\\ \texttt{je}\ \ell_1\\ \texttt{jmp}\ \ell_2\end{array}$$

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

Regarding the `return` statement, we recommend treating it as an assignment to the `rax` register followed by a jump to the conclusion of the `main` function.

Exercise 21 Expand your `select_instructions` pass to handle the new features of the $\mathcal{C}_{\mathsf{If}}$ language. Run the script to test your compiler on all the test programs.

4.9 Register Allocation

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

4.9.1 Liveness Analysis

Recall that for $\mathcal{L}_{\mathsf{Var}}$ we implemented liveness analysis for a single basic block (Section 3.2). With the addition of `if` expressions to $\mathcal{L}_{\mathsf{If}}$, `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 contraints 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 provide implementations of `topological_sort` and `transpose` in the file `graph.py` of the support code. 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 $\mathcal{L}_{\mathsf{If}}$ programs. However, in Chapter 5 we add loops to create $\mathcal{L}_{\mathsf{While}}$ and learn how to handle cycles in the control-flow graph.

The next question is how to analyze jump instructions. The locations that are live before a `jmp` should be the locations in $L_{\mathtt{before}}$ at the target of the jump. So we recommend maintaining a dictionary named `live_before_block` that maps each label to the $L_{\mathtt{before}}$ for the first instruction in its block. After performing liveness analysis on each block, we take the live-before set of its first instruction and associate that with the block's label in the `live_before_block` dictionary.

In $\mathrm{x86}^{\mathsf{Var}}_{\mathsf{If}}$ 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 `live_before_block`.

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 $\mathrm{x86}^{\mathsf{Var}}_{\mathsf{If}}$.

Exercise 22 Update the `uncover_live` function to perform liveness analysis, in reverse topological order, on all of the basic blocks in the program.

4.9.2 Build the Interference Graph

Many of the new instructions in $\mathrm{x86}^{\mathsf{Var}}_{\mathsf{If}}$ can be handled in the same way as the instructions in $\mathrm{x86}_{\mathsf{Var}}$. Some instructions, e.g., the `movzbq` instruction, require special care, similar to the `movq` instruction. See rule number 1 in Section 3.3.

Exercise 23 Update the `build_interference` pass for $\mathrm{x86}^{\mathsf{Var}}_{\mathsf{If}}$.

4.10 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 24 Update `patch_instructions` pass for $\mathrm{x86}^{\mathsf{Var}}_{\mathsf{If}}$.

4.11 Prelude and Conclusion

The generation of the `main` function with its prelude and conclusion must change to accomodate how the program now consists of one or more basic blocks. After the prelude in `main`, jump to the `start` block. Place the conclusion in a basic block labelled with `conclusion`.

Figure 4.14 shows a simple example program in $\mathcal{L}_{\mathsf{If}}$ 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 $\mathcal{L}_{\mathsf{If}}$.

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 4.7 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 `True` in `explicate_pred`, where we discard the `els` continuation.

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

```
print(42 if input_int() == 1 else 0)
```

⇓

```
start:
        tmp_0 = input_int()
        if tmp_0 == 1:
          goto block_3
        else:
          goto block_4
block_3:
        tmp_1 = 42
        goto block_2
block_4:
        tmp_1 = 0
        goto block_2
block_2:
        print(tmp_1)
        return 0
```

⇓

```
start:
        callq read_int
        movq %rax, tmp_0
        cmpq 1, tmp_0
        je block_3
        jmp block_4
block_3:
        movq 42, tmp_1
        jmp block_2
block_4:
        movq 0, tmp_1
        jmp block_2
block_2:
        movq tmp_1, %rdi
        callq print_int
        movq 0, %rax
        jmp conclusion
```

⇒

```
        .globl main
main:
        pushq %rbp
        movq %rsp, %rbp
        subq $0, %rsp
        jmp start
start:
        callq read_int
        movq %rax, %rcx
        cmpq $1, %rcx
        je block_3
        jmp block_4
block_3:
        movq $42, %rcx
        jmp block_2
block_4:
        movq $0, %rcx
        jmp block_2
block_2:
        movq %rcx, %rdi
        callq print_int
        movq $0, %rax
        jmp conclusion
conclusion:
        addq $0, %rsp
        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.

creation of a basic block until the point in time where we know it will be used. While Python does not provide direct support for lazy evaluation, it is easy to mimic. We can *delay* the evaluation of a computation by wrapping it inside a function with no parameters. We can *force* its evaluation by calling the function. However, in some cases of `explicate_pred`, etc., we will return a list of statements and in other cases we will return a function that computes a list of statements. We use the term *promise* to refer to a value that may be delayed. To uniformly deal with promises, we define the following `force` function that checks whether its input

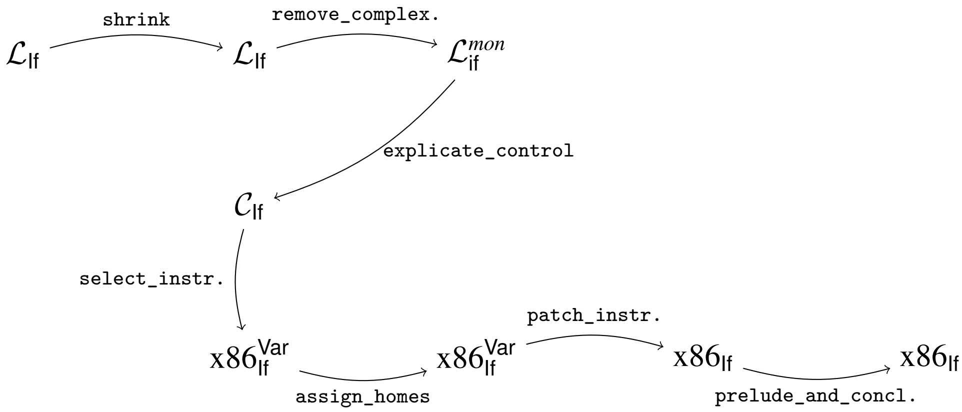


Figure 4.15
Diagram of the passes for $\mathcal{L}_{\mathsf{If}}$, a language with conditionals.

is delayed (i.e., whether it is a function) and then either 1) calls the function, or 2) returns the input.

```
def force(promise):
    if isinstance(promise, types.FunctionType):
        return promise()
    else:
        return promise
```

We use promises for the input and output of the functions `explicate_pred`, `explicate_assign`, `explicate_effect`, and `explicate_stmt`. So instead of taking and returning lists of statments, 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`. Here is the new version of the `create_block` auxiliary function that works on promises and that checks whether the block consists of a solitary `goto` statement.

```
def create_block(promise, basic_blocks):
  stmts = force(promise)
  match stmts:
    case [Goto(l)]:
       return Goto(l)
    case _:
      label = label_name(generate_name('block'))
      basic_blocks[label] = stmts
      return Goto(label)
```

```
x = input_int()
y = input_int()
print(y + 2          \
      if (x == 0      \
          if x < 1    \
          else x == 2) \
      else y + 10)
```

$\Rightarrow$

```
start:
    x = input_int()
    y = input_int()
    if x < 1:
        goto block_4
    else:
        goto block_5
block_4:
    if x == 0:
        goto block_2
    else:
        goto block_3
block_5:
    if x == 2:
        goto block_2
    else:
        goto block_3
block_2:
    tmp_0 = y + 2
    goto block_1
block_3:
    tmp_0 = y + 10
    goto block_1
block_1:
    print(tmp_0)
    return 0
```

Figure 4.16
Translation from $\mathcal{L}_{\mathsf{If}}$ to $\mathcal{C}_{\mathsf{If}}$ via the improved `explicate_control`.

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 25 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 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 26 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 $\text{x86}^{\mathsf{Var}}_{\mathsf{If}}$ to $\text{x86}^{\mathsf{Var}}_{\mathsf{If}}$. Run the script to test your compiler. Check

```
start:
        callq read_int
        movq %rax, tmp_0
        cmpq 1, tmp_0
        je block_3
        jmp block_4
block_3:
        movq 42, tmp_1
        jmp block_2
block_4:
        movq 0, tmp_1
        jmp block_2
block_2:
        movq tmp_1, %rdi
        callq print_int
        movq 0, %rax
        jmp conclusion
```

⇒

```
start:
        callq read_int
        movq %rax, tmp_0
        cmpq 1, tmp_0
        je block_3
        movq 0, tmp_1
        jmp block_2
block_3:
        movq 42, tmp_1
        jmp block_2
block_2:
        movq tmp_1, %rdi
        callq print_int
        movq 0, %rax
        jmp conclusion
```

Figure 4.17
Merging basic blocks by removing unnecessary jumps.

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 `explose-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 loops, one of the hallmarks of imperative programming languages. The following example demonstrates the `while` loop by computing the sum of the first five positive integers.

```
sum = 0
i = 5
while i > 0:
    sum = sum + i
    i = i - 1
print(sum)
```

The `while` loop consists of a condition expression and a body (a sequence of statements). The body is evaluated repeatedly so long as the condition remains true.

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

The concrete syntax of $\mathcal{L}_{\mathsf{While}}$ is defined in Figure 5.1 and its abstract syntax is defined in Figure 5.2. The definitional interpreter for $\mathcal{L}_{\mathsf{While}}$ is shown in Figure 5.3. We add a new case for `While` in the `interp_stmts` function, where we repeatedly interpret the `body` so long as the `test` expression remains true.

The type checker for $\mathcal{L}_{\mathsf{While}}$ is defined in Figure 5.4. A `while` loop is well typed if the type of the `test` expression is `bool` and the statements in the `body` are well typed.

At first glance, the translation of `while` loops to x86 seems straightforward because the $\mathcal{C}_{\mathsf{If}}$ intermediate language already supports `goto` and conditional branching. 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

$$
\begin{array}{rcl}
exp &::=& int \mid \texttt{input\_int()} \mid \texttt{-}exp \mid exp \texttt{+} exp \mid exp \texttt{-} exp \mid \texttt{(}exp\texttt{)} \\
stmt &::=& \texttt{print(}exp\texttt{)} \mid exp \\
\hline
exp &::=& var \\
stmt &::=& var \texttt{ = } exp \\
\hline
cmp &::=& \texttt{==} \mid \texttt{!=} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \\
exp &::=& \texttt{True} \mid \texttt{False} \mid exp \texttt{ and } exp \mid exp \texttt{ or } exp \mid \texttt{not } exp \\
&\mid& exp\ cmp\ exp \mid exp \texttt{ if } exp \texttt{ else } exp \\
stmt &::=& \texttt{if } exp\texttt{: } stmt^{+} \texttt{ else: } stmt^{+} \\
\hline
stmt &::=& \texttt{while } exp\texttt{: } stmt^{+} \\
\mathcal{L}_{\mathsf{While}} &::=& stmt^{*}
\end{array}
$$

Figure 5.1
The concrete syntax of $\mathcal{L}_{\mathsf{While}}$, extending $\mathcal{L}_{\mathsf{If}}$ (Figure 4.1).

$$
\begin{array}{rcl}
binaryop &::=& \texttt{Add()} \mid \texttt{Sub()} \\
unaryop &::=& \texttt{USub()} \\
exp &::=& \texttt{Constant(}int\texttt{)} \mid \texttt{Call(Name('input\_int'),[])} \\
&\mid& \texttt{UnaryOp(}unaryop\texttt{,}exp\texttt{)} \mid \texttt{BinOp(}binaryop\texttt{,}exp\texttt{,}exp\texttt{)} \\
stmt &::=& \texttt{Expr(Call(Name('print'),[}exp\texttt{]))} \mid \texttt{Expr(}exp\texttt{)} \\
\hline
exp &::=& \texttt{Name(}var\texttt{)} \\
stmt &::=& \texttt{Assign([Name(}var\texttt{)], }exp\texttt{)} \\
\hline
boolop &::=& \texttt{And()} \mid \texttt{Or()} \\
unaryop &::=& \texttt{Not()} \\
cmp &::=& \texttt{Eq()} \mid \texttt{NotEq()} \mid \texttt{Lt()} \mid \texttt{LtE()} \mid \texttt{Gt()} \mid \texttt{GtE()} \\
bool &::=& \texttt{True} \mid \texttt{False} \\
exp &::=& \texttt{Constant(}bool\texttt{)} \mid \texttt{BoolOp(}boolop\texttt{,[}exp\texttt{,}exp\texttt{])} \\
&\mid& \texttt{Compare(}exp\texttt{,[}cmp\texttt{],[}exp\texttt{])} \mid \texttt{IfExp(}exp\texttt{,}exp\texttt{,}exp\texttt{)} \\
stmt &::=& \texttt{If(}exp\texttt{, }stmt^{+}\texttt{, }stmt^{+}\texttt{)} \\
\hline
stmt &::=& \texttt{While(}exp\texttt{, }stmt^{+}\texttt{, [])} \\
\mathcal{L}_{\mathsf{While}} &::=& \texttt{Module(}stmt^{*}\texttt{)}
\end{array}
$$

Figure 5.2
The abstract syntax of $\mathcal{L}_{\mathsf{While}}$, extending $\mathcal{L}_{\mathsf{If}}$ (Figure 4.2).

```
class InterpLwhile(InterpLif):
  def interp_stmts(self, ss, env):
    if len(ss) == 0:
      return
    match ss[0]:
      case While(test, body, []):
        while self.interp_exp(test, env):
            self.interp_stmts(body, env)
        return self.interp_stmts(ss[1:], env)
      case _:
        return super().interp_stmts(ss, env)
```

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

```
class TypeCheckLwhile(TypeCheckLif):

  def type_check_stmts(self, ss, env):
    if len(ss) == 0:
      return
    match ss[0]:
      case While(test, body, []):
        test_t = self.type_check_exp(test, env)
        check_type_equal(bool, test_t, test)
        body_t = self.type_check_stmts(body, env)
        return self.type_check_stmts(ss[1:], env)
      case _:
        return super().type_check_stmts(ss, env)
```

Figure 5.4
Type checker for the $\mathcal{L}_{\mathsf{While}}$ language.

this we analyzed the control-flow graph in reverse topological order (Section 4.9.1), but topological order is only well-defined for acyclic graphs.

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.

```
mainstart:
        movq $0, sum
        movq $5, i
        jmp block5
block5:
        cmpq $0, i
        jg block7
        jmp block8
block7:
        addq i, sum
        subq $1, i
        jmp block5
block8:
        movq sum, %rdi
        callq print_int
        movq $0, %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 $\sqcup$.[5] 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.

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

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 $\bot_M$ that sends every label to the empty set, i.e., $\bot_M(\ell) = \emptyset$.

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 $\bot$ and iterating f as follows.

$$\bot \sqsubseteq f(\bot) \sqsubseteq f(f(\bot)) \sqsubseteq \cdots \sqsubseteq f^n(\bot) \sqsubseteq \cdots$$

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.

$$\bot \sqsubseteq f(\bot) \sqsubseteq f(f(\bot)) \sqsubseteq \cdots \sqsubseteq f^k(\bot) = f^{k+1}(\bot) = m_s$$

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

```
def analyze_dataflow(G, transfer, bottom, join):
    trans_G = transpose(G)
    mapping = dict((v, bottom) for v in G.vertices())
    worklist = deque(G.vertices)
    while worklist:
        node = worklist.pop()
        input = reduce(join, [mapping[v] for v in trans_G.adjacent(node)], bottom)
        output = transfer(node, input)
        if output != mapping[node]:
            mapping[node] = output
            worklist.extend(G.adjacent(node))
```

Figure 5.5
Generic work list algorithm for dataflow analysis

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.

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

5.3 Remove Complex Operands

The change needed for this pass is to add a case for the `while` statement. The condition of a `while` loop is allowed to be a complex expression, just like the condition of the `if` statement. Figure 5.6 defines the output language $\mathcal{L}^{mon}_{\mathsf{While}}$ of this pass.

5.4 Explicate Control

The output of this pass is the language $\mathcal{C}_{\mathsf{If}}$. No new language features are needed in the output because a `while` loop can be expressed in terms of `goto` and `if` statements, which are already in $\mathcal{C}_{\mathsf{If}}$. Add a case for the `while` statement to the `explicate_stmt` method, using `explicate_pred` to process the condition expression.

$$
\begin{array}{rcl}
\mathit{atm} & ::= & \texttt{Constant(}\mathit{int}\texttt{)} \mid \texttt{Name(}\mathit{var}\texttt{)} \\
\mathit{exp} & ::= & \mathit{atm} \mid \texttt{Call(Name('input\_int'),[])} \\
 & \mid & \texttt{UnaryOp(}\mathit{unaryop}\texttt{,}\mathit{atm}\texttt{)} \mid \texttt{BinOp(}\mathit{atm}\texttt{,}\mathit{binaryop}\texttt{,}\mathit{atm}\texttt{)} \\
\mathit{stmt} & ::= & \texttt{Expr(Call(Name('print'),[}\mathit{atm}\texttt{]))} \mid \texttt{Expr(}\mathit{exp}\texttt{)} \\
 & \mid & \texttt{Assign([Name(}\mathit{var}\texttt{)], }\mathit{exp}\texttt{)} \\
\hline
\mathit{atm} & ::= & \texttt{Constant(}\mathit{bool}\texttt{)} \\
\mathit{exp} & ::= & \texttt{Compare(}\mathit{atm}\texttt{,[}\mathit{cmp}\texttt{],[}\mathit{atm}\texttt{])} \mid \texttt{IfExp(}\mathit{exp}\texttt{,}\mathit{exp}\texttt{,}\mathit{exp}\texttt{)} \\
 & \mid & \texttt{Begin(}\mathit{stmt}^{*}\texttt{, }\mathit{exp}\texttt{)} \\
\mathit{stmt} & ::= & \texttt{If(}\mathit{exp}\texttt{, }\mathit{stmt}^{*}\texttt{, }\mathit{stmt}^{*}\texttt{)} \\
\hline
\mathit{stmt} & ::= & \texttt{While(}\mathit{exp}\texttt{, }\mathit{stmt}^{+}\texttt{, [])} \\
\mathcal{L}^{mon}_{\mathsf{While}} & ::= & \texttt{Module(}\mathit{stmt}^{*}\texttt{)}
\end{array}
$$

Figure 5.6
$\mathcal{L}^{mon}_{\mathsf{While}}$ is $\mathcal{L}_{\mathsf{While}}$ in monadic normal form.

5.5 Register Allocation

As discussed in Section 5.2, the presence of loops in $\mathcal{L}_{\mathsf{While}}$ means that the control-flow graphs may contain cycles, which complicates the liveness analysis needed for register allocation.

5.5.1 Liveness Analysis

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 4.9.1).

The `analyze_dataflow` function has four parameters.

1. The first parameter `G` should be a directed graph from the `graph.py` file in the support code that represents the control-flow graph.
2. The second parameter `transfer` is 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 live-before and live-after sets 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. The bottom of the lattice is the empty set and the join operator is set union.

Figure 5.7 provides an overview of all the passes needed for the compilation of $\mathcal{L}_{\mathsf{While}}$.

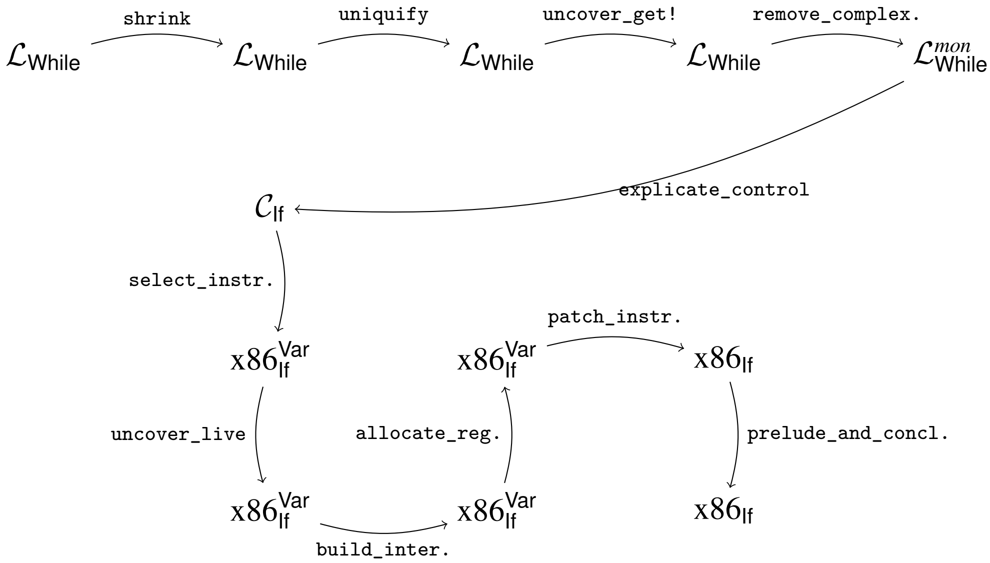


Figure 5.7
Diagram of the passes for $\mathcal{L}_{\mathsf{While}}$.

6 Tuples and Garbage Collection

In this chapter we study the implementation of tuples. 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 $\mathcal{L}_{\mathsf{Tup}}$ language including its interpreter and type checker. The $\mathcal{L}_{\mathsf{Tup}}$ language extends the $\mathcal{L}_{\mathsf{While}}$ 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 6.3 through 6.8 discuss the necessary changes and additions to the compiler passes, including a new compiler pass named `expose_allocation`.

6.1 The $\mathcal{L}_{\mathsf{Tup}}$ Language

Figure 6.1 defines the concrete syntax for $\mathcal{L}_{\mathsf{Tup}}$ and Figure 6.2 defines the abstract syntax. The $\mathcal{L}_{\mathsf{Tup}}$ language adds 1) tuple creation via a comma-separated list of expressions, 2) accessing an element of a tuple with the square bracket notation, i.e., `t[n]` returns the element at index `n` of tuple `t`, 3) the `is` comparison operator, and 4) obtaining the number of elements (the length) of a tuple. In this chapter, we restrict access indices to constant integers. The program below shows an example use of tuples. It creates a tuple `t` containing the elements `40`, `True`, and another tuple that contains just `2`. The element at index 1 of `t` is `True`, 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`.

```
t = 40, True, (2,)
print( t[0] + t[2][0] if t[1] else 44 )
```

Tuples raise several interesting new issues. First, variable binding performs a shallow-copy when dealing with tuples, which means that different variables can refer to the same tuple, that is, two variables can be *aliases* for the same entity. Consider the following example in which both `t1` and `t2` refer to the same tuple

$$
\begin{array}{rcl}
exp & ::= & int \mid \texttt{input\_int()} \mid \texttt{-}exp \mid exp\,\texttt{+}\,exp \mid exp\,\texttt{-}\,exp \mid \texttt{(}exp\texttt{)} \\
stmt & ::= & \texttt{print(}exp\texttt{)} \mid exp \\
\hline
exp & ::= & var \\
stmt & ::= & var\,\texttt{=}\,exp \\
\hline
cmp & ::= & \texttt{==} \mid \texttt{!=} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \\
exp & ::= & \texttt{True} \mid \texttt{False} \mid exp\ \texttt{and}\ exp \mid exp\ \texttt{or}\ exp \mid \texttt{not}\ exp \\
 & \mid & exp\ cmp\ exp \mid exp\ \texttt{if}\ exp\ \texttt{else}\ exp \\
stmt & ::= & \texttt{if}\ exp\texttt{:}\ stmt^{+}\ \texttt{else:}\ stmt^{+} \\
\hline
stmt & ::= & \texttt{while}\ exp\texttt{:}\ stmt^{+} \\
\hline
cmp & ::= & \texttt{is} \\
exp & ::= & exp\texttt{,}\ \ldots\ \texttt{,}exp \mid exp\texttt{[}int\texttt{]} \mid \texttt{len(}exp\texttt{)} \\
\mathcal{L}_{\mathsf{Tup}} & ::= & stmt^{*}
\end{array}
$$

Figure 6.1
The concrete syntax of $\mathcal{L}_{\mathsf{Tup}}$, extending $\mathcal{L}_{\mathsf{While}}$ (Figure 5.1).

$$
\begin{array}{rcl}
binaryop & ::= & \texttt{Add()} \mid \texttt{Sub()} \\
unaryop & ::= & \texttt{USub()} \\
exp & ::= & \texttt{Constant(}int\texttt{)} \mid \texttt{Call(Name('input\_int'),[])} \\
 & \mid & \texttt{UnaryOp(}unaryop\texttt{,}exp\texttt{)} \mid \texttt{BinOp(}binaryop\texttt{,}exp\texttt{,}exp\texttt{)} \\
stmt & ::= & \texttt{Expr(Call(Name('print'),[}exp\texttt{]))} \mid \texttt{Expr(}exp\texttt{)} \\
\hline
exp & ::= & \texttt{Name(}var\texttt{)} \\
stmt & ::= & \texttt{Assign([Name(}var\texttt{)],}\ exp\texttt{)} \\
\hline
boolop & ::= & \texttt{And()} \mid \texttt{Or()} \\
unaryop & ::= & \texttt{Not()} \\
cmp & ::= & \texttt{Eq()} \mid \texttt{NotEq()} \mid \texttt{Lt()} \mid \texttt{LtE()} \mid \texttt{Gt()} \mid \texttt{GtE()} \\
bool & ::= & \texttt{True} \mid \texttt{False} \\
exp & ::= & \texttt{Constant(}bool\texttt{)} \mid \texttt{BoolOp(}boolop\texttt{,[}exp\texttt{,}exp\texttt{])} \\
 & \mid & \texttt{Compare(}exp\texttt{,[}cmp\texttt{],[}exp\texttt{])} \mid \texttt{IfExp(}exp\texttt{,}exp\texttt{,}exp\texttt{)} \\
stmt & ::= & \texttt{If(}exp\texttt{,}\ stmt^{+}\texttt{,}\ stmt^{+}\texttt{)} \\
\hline
stmt & ::= & \texttt{While(}exp\texttt{,}\ stmt^{+}\texttt{,}\ \texttt{[])} \\
\hline
cmp & ::= & \texttt{Is()} \\
exp & ::= & \texttt{Tuple(}exp^{+}\texttt{,Load())} \mid \texttt{Subscript(}exp\texttt{,Constant(}int\texttt{),Load())} \\
 & \mid & \texttt{Call(Name('len'),[}exp\texttt{])} \\
\mathcal{L}_{\mathsf{While}} & ::= & \texttt{Module(}stmt^{*}\texttt{)}
\end{array}
$$

Figure 6.2
The abstract syntax of $\mathcal{L}_{\mathsf{Tup}}$.

value but t3 refers to a different tuple value but with equal elements. The result of the program is 42.

```
t1 = 3, 7
t2 = t1
t3 = 3, 7
print( 42 if (t1 is t2) and not (t1 is t3) else 0 )
```

The next issue concerns the lifetime of tuples. When does their lifetime end? Notice that $\mathcal{L}_{\mathsf{Tup}}$ 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 x goes out of scope when the function returns, prior to reading the tuple element at index zero. (We study the compilation of functions in Chapter 7.)

```
def f():
    x = 42, 43
    return x
t = f()
print( t[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 $\mathcal{L}_{\mathsf{Tup}}$ language. We represent tuples with Python lists in the interpreter because we need to write to them (Section 6.3). (Python tuples are immutable.) We define element access, the is operator, and the len operator for $\mathcal{L}_{\mathsf{Tup}}$ in terms of the corresponding operations in Python.

Figure 6.4 shows the type checker for $\mathcal{L}_{\mathsf{Tup}}$, 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 records the type of each tuple expression in a new field named has_type. Because the type checker has to compute the type of each tuple access, the index must be a constant.

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.[6]

6. The term "object" as used in the context of object-oriented programming has a more specific meaning than how we are using the term here.

```
class InterpLtup(InterpLwhile):
  def interp_cmp(self, cmp):
    match cmp:
      case Is():
        return lambda x, y: x is y
      case _:
        return super().interp_cmp(cmp)
  def interp_exp(self, e, env):
    match e:
      case Tuple(es, Load()):
        return tuple([self.interp_exp(e, env) for e in es])
      case Subscript(tup, index, Load()):
        t = self.interp_exp(tup, env)
        n = self.interp_exp(index, env)
        return t[n]
      case _:
        return super().interp_exp(e, env)
```

Figure 6.3
Interpreter for the $\mathcal{L}_{\mathsf{Tup}}$ language.

```
class TypeCheckLtup(TypeCheckLwhile):
  def type_check_exp(self, e, env):
    match e:
      case Compare(left, [cmp], [right]) if isinstance(cmp, Is):
        l = self.type_check_exp(left, env)
        r = self.type_check_exp(right, env)
        check_type_equal(l, r, e)
        return bool
      case Tuple(es, Load()):
        ts = [self.type_check_exp(e, env) for e in es]
        e.has_type = tuple(ts)
        return e.has_type
      case Subscript(tup, Constant(index), Load()):
        tup_ty = self.type_check_exp(tup, env)
        index_ty = self.type_check_exp(Constant(index), env)
        check_type_equal(index_ty, int, index)
        match tup_ty:
          case tuple(ts):
            return ts[index]
          case _:
            raise Exception('error: expected a tuple, not ' + repr(tup_ty))
      case _:
        return super().type_check_exp(e, env)
```

Figure 6.4
Type checker for the $\mathcal{L}_{\mathsf{Tup}}$ language.

Unfortunately, it is impossible to know precisely which objects will be accessed in the future and which will not. Instead, garbage collectors overapproximate 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 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 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, there are three pointers in the root set, one in a register and two on the stack. All of the live objects have been copied to the ToSpace (the right-hand side of Figure 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 $\mathcal{L}_{\mathsf{Tup}}$ because it contains a cycle. However, creating cycles will be possible once we get to $\mathcal{L}_{\mathsf{Dyn}}$. We design the garbage collector to deal with cycles to begin with so we will not need to revisit this issue.

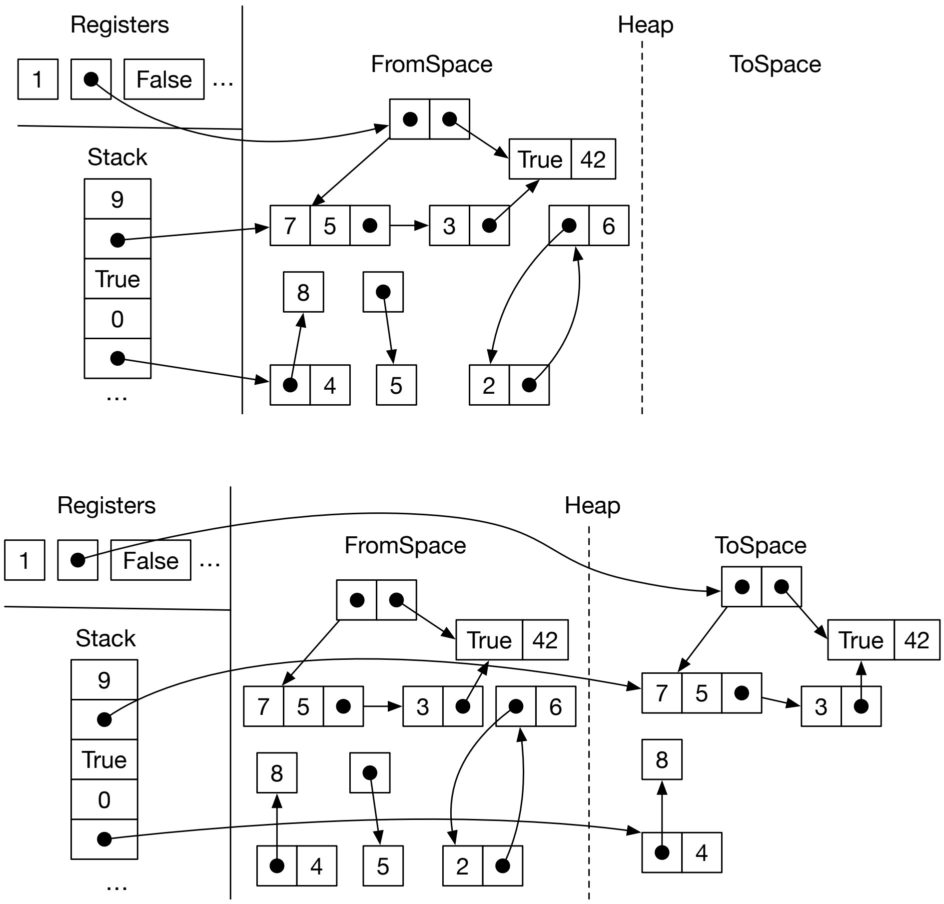


Figure 6.5
A copying collector in action.

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 tuples that are immediately reachable from the root set into the ToSpace to form the initial queue. When we copy a tuple, we mark the old tuple to indicate that it has been visited. We discuss how this marking is accomplish in Section 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 tuple that are directly reachable from it to the ToSpace, placing them at the back of the queue. The algorithm then updates the pointers in the popped tuple so they point to the newly copied tuples.

Getting back to Figure 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 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 generate type-specific code for collecting or to generate tables that can guide the collector (Appel 1989; Goldberg 1991; Diwan, Moss, and Hudson 1992).

Dynamically typed languages, such as Python, need to tag objects anyways, so option 1 is a natural choice for those languages. However, $\mathcal{L}_{\mathsf{Tup}}$ 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 `TupleType`, 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

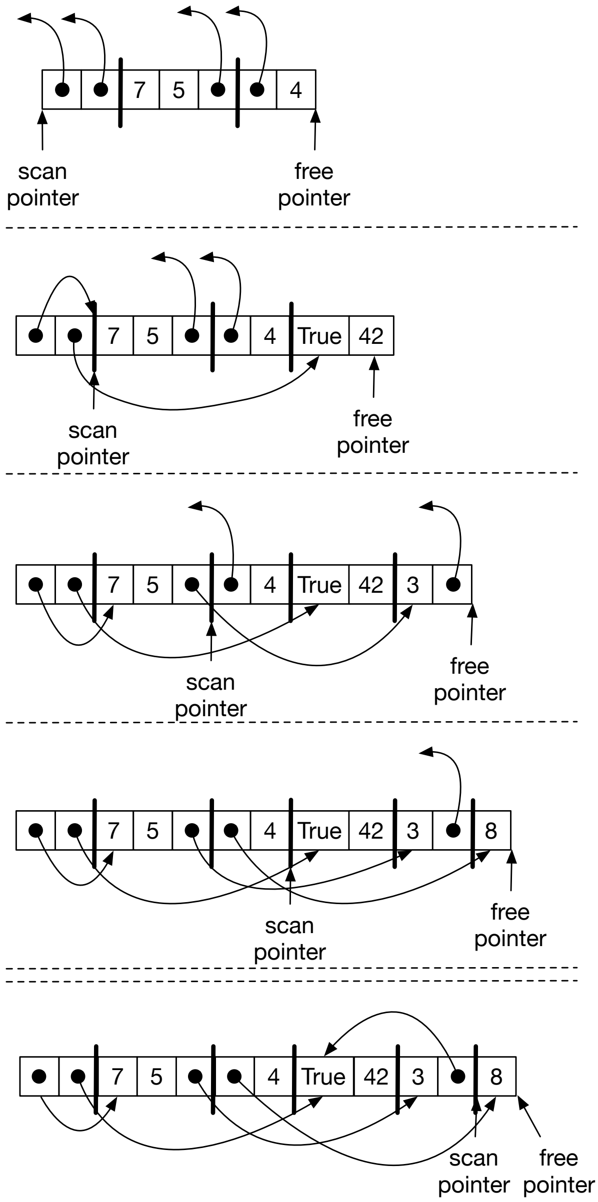


Figure 6.6
Depiction of the Cheney algorithm copying the live tuples.

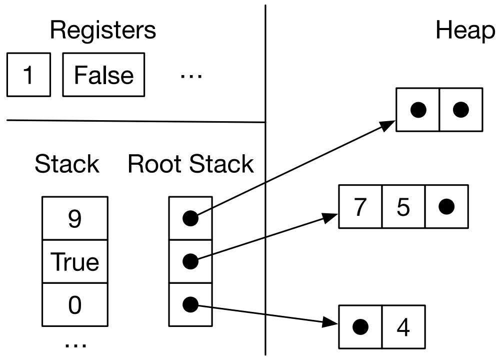


Figure 6.7
Maintaining a root stack to facilitate garbage collection.

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 50 elements, so we just need 50 bits for the pointer mask.[7] 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

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

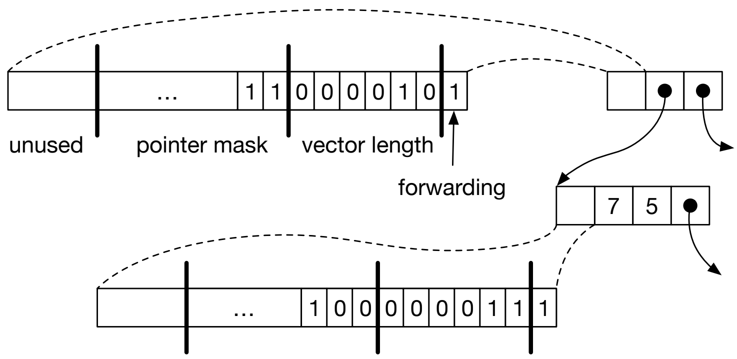


Figure 6.8
Representation of tuples in the heap.

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

Figure 6.9
The compiler's interface to the garbage collector.

should be called in the prelude of the `main` function. The arguments of `initialize` 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 FromSpace is the difference between the `fromspace_end` and the `free_ptr`. The `collect` function should be called when there is not enough room left in the FromSpace for the next allocation. The `collect` function takes a pointer to the current top of the root stack (one past the last item that was pushed) and the number of bytes that need to be allocated. The `collect` function performs the copying collection and leaves the heap in a state such that the next allocation will succeed.

The introduction of garbage collection has a non-trivial impact on our compiler passes. We introduce a new compiler pass named `expose_allocation`. We 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.

```
print( ((42,),)[0][0] )
```

6.3 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 the code generated by `expose_allocation` contains complex operands.

The output of `expose_allocation` is a language $\mathcal{L}_{\mathsf{Alloc}}$ that extends $\mathcal{L}_{\mathsf{Tup}}$ with new forms that we use in the translation of tuple creation.

$$
\begin{array}{lcl}
\mathit{exp} & ::= & \cdots \\
 & \mid & \texttt{collect}(\mathit{int}) \mid \texttt{allocate}(\mathit{int}, \mathit{type}) \mid \texttt{global\_value}(\mathit{name}) \\
 & \mid & \texttt{begin:}\ \mathit{stmt}^*\ \mathit{exp} \\
\mathit{stmt} & ::= & \mathit{exp}[\mathit{int}]\ \texttt{=}\ \mathit{exp}
\end{array}
$$

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: `TupleType([`$\mathit{type}_1, \ldots, \mathit{type}_n$ `])` where type_i is the type of the ith element in the tuple. The `global_value(`name`)` form reads the value of a global variable, such as `free_ptr`. The `begin` form is an expression that executes a sequence of statements and then produces the value of the expression at the end.

The following shows the transformation of tuple creation into 1) a sequence of temporary variables 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. The *type* needed for the second argument of the `allocate` form can be obtained from the `has_type` field of the tuple AST node, which is stored there by running the type checker for $\mathcal{L}_{\mathsf{Tup}}$ immediately before this pass.

```
(e0, ..., en-1)
⟹
begin:
    x0 = e0
       ⋮
    xn-1 = en-1
    if global_value(free_ptr) + bytes < global_value(fromspace_end):
        0
    else:
        collect(bytes)
    v = allocate(len, type)
    v[0] = x0
       ⋮
    v[n-1] = xn-1
    v
```

The sequencing of the initializing expressions $e_0, \ldots, e_{n-1}$ 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.

6.4 Remove Complex Operands

The expressions `allocate`, `global_value`, `begin`, and tuple access should be treated as complex operands. The sub-expressions of tuple access must be atomic. Figure 6.11 shows the grammar for the output language $\mathcal{L}^{mon}_{\mathsf{Alloc}}$ of this pass, which is $\mathcal{L}_{\mathsf{Alloc}}$ in monadic normal form.

6.5 Explicate Control and the $\mathcal{C}_{\mathsf{Tup}}$ language

The output of `explicate_control` is a program in the intermediate language $\mathcal{C}_{\mathsf{Tup}}$, whose abstract syntax is defined in Figure 6.12. The new expressions of $\mathcal{C}_{\mathsf{Tup}}$ include `allocate`, accessing tuple elements, and `global_value`. $\mathcal{C}_{\mathsf{Tup}}$ also includes the `collect` statement and assignment to a tuple element. The `explicate_control` pass can treat these new forms much like the other forms that we've already encoutered.

6.6 Select Instructions and the $\text{x86}_{\mathsf{Global}}$ 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).

```
print( T1[0][0] )
```

where T_1 is

```
begin:
      tmp.1 = T2
      if global_value(free_ptr) + 16 < global_value(fromspace_end):
         0
      else:
         collect(16)
      tmp.2 = allocate(1, TupleType(TupleType([int])))
      tmp.2[0] = tmp.1
      tmp.2
```

and T_2 is

```
begin:
      tmp.3 = 42
      if global_value(free_ptr) + 16 < global_value(fromspace_end):
         0
      else:
         collect(16)
      tmp.4 = allocate(1, TupleType([int]))
      tmp.4[0] = tmp.3
      tmp.4
```

Figure 6.10
Output of the `expose_allocation` pass.

atm	::=	`Constant(`*int*`)` | `Name(`*var*`)`
exp	::=	*atm* | `Call(Name('input_int'),[])`
	|	`UnaryOp(`*unaryop*`,`*atm*`)` | `BinOp(`*atm*`,`*binaryop*`,`*atm*`)`
stmt	::=	`Expr(Call(Name('print'),[`*atm*`]))` | `Expr(`*exp*`)`
	|	`Assign([Name(`*var*`)], `*exp*`)`
atm	::=	`Constant(`*bool*`)`
exp	::=	`Compare(`*atm*`,[`*cmp*`],[`*atm*`])` | `IfExp(`*exp*`,`*exp*`,`*exp*`)`
	|	`Begin(`$stmt^*$`, `*exp*`)`
stmt	::=	`If(`*exp*`, `$stmt^*$`, `$stmt^*$`)`
stmt	::=	`While(`*exp*`, `$stmt^+$`, [])`
exp	::=	`Subscript(`*atm*`,`*atm*`,Load())`
	|	`Call(Name('len'),[`*atm*`])`
	|	`Allocate(`*int*`, `*type*`)` | `GlobalValue(`*var*`)`
stmt	::=	`Assign([Subscript(`*atm*`,`*atm*`,Store())], `*atm*`)`
	|	`Collect(`*int*`)`
$\mathcal{L}^{mon}_{\mathsf{Alloc}}$	::=	`Module(`$stmt^*$`)`

Figure 6.11
$\mathcal{L}^{mon}_{\mathsf{Alloc}}$ is $\mathcal{L}_{\mathsf{Alloc}}$ in monadic normal form.

$$
\begin{array}{lcl}
\mathit{atm} & ::= & \texttt{Constant(}\mathit{int}\texttt{)} \mid \texttt{Name(}\mathit{var}\texttt{)} \mid \texttt{Constant(}\mathit{bool}\texttt{)} \\
\mathit{exp} & ::= & \mathit{atm} \mid \texttt{Call(Name('input\_int'),[])} \\
 & \mid & \texttt{BinOp(}\mathit{atm}\texttt{,}\mathit{binaryop}\texttt{,}\mathit{atm}\texttt{)} \mid \texttt{UnaryOp(}\mathit{unaryop}\texttt{,}\mathit{atm}\texttt{)} \\
 & \mid & \texttt{Compare(}\mathit{atm}\texttt{,[}\mathit{cmp}\texttt{],[}\mathit{atm}\texttt{])} \\
\mathit{stmt} & ::= & \texttt{Expr(Call(Name('print'),[}\mathit{atm}\texttt{]))} \mid \texttt{Expr(}\mathit{exp}\texttt{)} \\
 & \mid & \texttt{Assign([Name(}\mathit{var}\texttt{)], }\mathit{exp}\texttt{)} \mid \texttt{Return(}\mathit{exp}\texttt{)} \mid \texttt{Goto(}\mathit{label}\texttt{)} \\
 & \mid & \texttt{If(Compare(}\mathit{atm}\texttt{,[}\mathit{cmp}\texttt{],[}\mathit{atm}\texttt{]), [Goto(}\mathit{label}\texttt{)], [Goto(}\mathit{label}\texttt{)])} \\
\hline
\mathit{exp} & ::= & \texttt{Subscript(}\mathit{atm}\texttt{,}\mathit{atm}\texttt{,Load())} \mid \texttt{Allocate(}\mathit{int}, \mathit{type}\texttt{)} \\
 & \mid & \texttt{GlobalValue(}\mathit{var}\texttt{)} \mid \texttt{Call(Name('len'),[}\mathit{atm}\texttt{])} \\
\mathit{stmt} & ::= & \texttt{Collect(}\mathit{int}\texttt{)} \\
 & \mid & \texttt{Assign([Subscript(}\mathit{atm}\texttt{,}\mathit{atm}\texttt{,Store())], }\mathit{atm}\texttt{)} \\
\mathcal{C}_{\mathsf{Tup}} & ::= & \texttt{CProgram(\{}\mathit{label}\texttt{: }\mathit{stmt}^{*}\texttt{, }\ldots\texttt{\})}
\end{array}
$$

Figure 6.12
The abstract syntax of $\mathcal{C}_{\mathsf{Tup}}$, extending $\mathcal{C}_{\mathsf{If}}$ (Figure 4.8).

The tuple read and write forms translate into `movq` instructions. (The plus one in the offset is to get past the tag at the beginning of the tuple representation.)

```
lhs = tup[n]
⟹
movq tup′, %r11
movq 8(n+1)(%r11), lhs′

tup[n] = rhs
⟹
movq tup′, %r11
movq rhs′, 8(n+1)(%r11)
```

The tup' and rhs' are obtained by translating from $\mathcal{C}_{\mathsf{Tup}}$ 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 `len` 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 bitwise-or operator `|` and the shift-left operator `«` 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 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, TupleType([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 $\text{x86}_{\text{Global}}$ language is defined in Figures 6.13 and 6.14. It differs from x86_{If} just in the addition of global variables. Figure 6.15 shows the output of the `select_instructions` pass on the running example.

$$
\begin{array}{lcl}
\mathit{arg} & ::= & \$\mathit{int} \mid \%\mathit{reg} \mid \mathit{int}(\%\mathit{reg}) \mid \%\mathit{bytereg} \mid \mathit{var}(\texttt{\%rip}) \\
\text{x86}_{\text{Global}} & ::= & \texttt{.globl main} \\
 & & \texttt{main:}\ \mathit{instr}^{*}
\end{array}
$$

Figure 6.13
The concrete syntax of $\text{x86}_{\text{Global}}$ (extends x86_{If} of Figure 4.9).

$$
\begin{array}{lcl}
\mathit{arg} & ::= & \texttt{Constant}(\mathit{int}) \mid \texttt{Reg}(\mathit{reg}) \mid \texttt{Deref}(\mathit{reg},\mathit{int}) \mid \texttt{ByteReg}(\mathit{reg}) \\
 & \mid & \texttt{Global}(\mathit{var}) \\
\text{x86}_{\text{Global}} & ::= & \texttt{X86Program}(((\mathit{label}\ .\ \mathit{block})\ \ldots))
\end{array}
$$

Figure 6.14
The abstract syntax of $\text{x86}_{\text{Global}}$ (extends x86_{If} of Figure 4.10).

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

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

Figure 6.15
Output of the `select_instructions` pass.

6.7 Register Allocation

As discussed earlier in this chapter, the garbage collector needs to access all the pointers in the root set, that is, all variables that are 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 generated by the type checker for $\mathcal{C}_{\mathsf{Tup}}$, stored a field named `var_types` in the `CProgram` AST mode. You'll need to propagate that information so that it is available in this pass.

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 `CProgram` output of this pass changes to also record the number of spills to the root stack.

6.8 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 decrement `r15` by 8 bytes.

One issue that deserves special care is that there may be a call to `collect` prior to the initializing assignments for all the variables in the root stack. We do not want the garbage collector to accidentally think that some uninitialized variable is a pointer that needs to be followed. Thus, we zero-out all locations on the root stack in the prelude of `main`. In Figure 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 $\mathcal{L}_{\mathsf{Tup}}$.

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

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

        .globl main
main:
        pushq  %rbp
        movq   %rsp, %rbp
        pushq  %r13
        pushq  %r12
        pushq  %rbx
        pushq  %r14
        subq   $0, %rsp
        movq $16384, %rdi
        movq $16384, %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   %r14
        popq   %rbx
        popq   %r12
        popq   %r13
        popq   %rbp
        retq
```

Figure 6.16

Output of the `prelude_and_conclusion` pass.

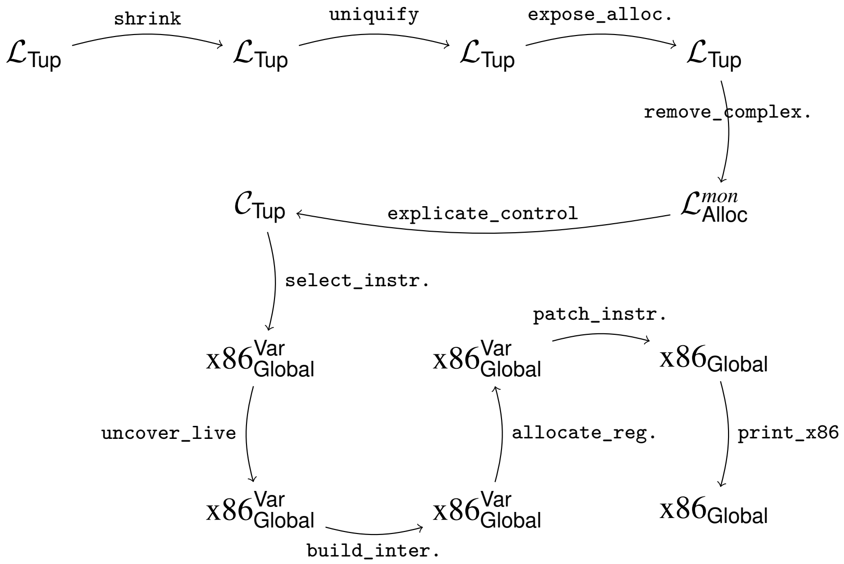


Figure 6.17
Diagram of the passes for $\mathcal{L}_{\mathsf{Tup}}$, a language with tuples.

6.9 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 its comes to garbage collection, such as mark-and-sweep (McCarthy 1960) and reference counting (Collins 1960). The strengths of copying collectors are that allocation is fast (just a comparison and pointer increment), there is no fragmentation, cyclic garbage is collected, and the time complexity of collection only depends on the amount of live data, and not on the amount of garbage (Wilson 1992). The main disadvantages of a two-space copying collector is that it uses a lot of extra space and takes a long time to perform the copy, though these problems are ameliorated in generational collectors. Object-oriented programs tend to allocate many small objects and generate a lot of garbage, so copying and generational collectors are a good fit (Dieckmann and Hölzle 1999). 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 Python in which only top-level function definitions are allowed.. This kind of function is a realistic example as the C language imposes similar restrictions. It is also 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 $\mathcal{L}_{\mathsf{Fun}}$ 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 $\mathcal{L}_{\mathsf{Fun}}$ language. Programs in $\mathcal{L}_{\mathsf{Fun}}$ begin with zero or more function definitions. The function names from these definitions are in-scope for the entire program, including all other function definitions (so the ordering of function definitions does not matter). The abstract syntax for function parameters in Figure 7.2 is a list of pairs, where each pair consists of a parameter name and its type. This design differs from Python's `ast` module, which has a more complex structure for function parameters to handle keyword parameters, defaults, and so on. The type checker in `type_check_Lfun` converts the complex Python abstract syntax into the simpler syntax of Figure 7.2. The fourth and sixth parameters of the `FunctionDef` constructor are for decorators and a type comment, neither of which are used by our compiler. We recommend replacing them with `None` in the `shrink` pass. 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 `Call(`*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

`Callable[[`$type_1$`,`$\cdots$`,`$type_n$`], `$type_R$`]`

for a function whose n parameters have the types $type_1$ through $type_n$ and whose return type is $type_R$. The main limitation of these functions (with respect to Python functions) is that they are not lexically scoped. That is, the only external entities

$$
\begin{array}{rcl}
exp & ::= & int \mid \texttt{input\_int()} \mid \texttt{-}exp \mid exp \texttt{+} exp \mid exp \texttt{-} exp \mid \texttt{(}exp\texttt{)} \\
stmt & ::= & \texttt{print(}exp\texttt{)} \mid exp \\
\hline
exp & ::= & var \\
stmt & ::= & var \texttt{=} exp \\
\hline
cmp & ::= & \texttt{==} \mid \texttt{!=} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \\
exp & ::= & \texttt{True} \mid \texttt{False} \mid exp\ \texttt{and}\ exp \mid exp\ \texttt{or}\ exp \mid \texttt{not}\ exp \\
 & \mid & exp\ cmp\ exp \mid exp\ \texttt{if}\ exp\ \texttt{else}\ exp \\
stmt & ::= & \texttt{if}\ exp\texttt{:}\ stmt^{+}\ \texttt{else:}\ stmt^{+} \\
\hline
stmt & ::= & \texttt{while}\ exp\texttt{:}\ stmt^{+} \\
\hline
cmp & ::= & \texttt{is} \\
exp & ::= & exp\texttt{,}\ldots\texttt{,}exp \mid exp\texttt{[}int\texttt{]} \mid \texttt{len(}exp\texttt{)} \\
\hline
type & ::= & \texttt{int} \mid \texttt{bool} \mid \texttt{tuple[}type^{+}\texttt{]} \mid \texttt{Callable[[}type\texttt{,}\ldots\texttt{],}\ type\texttt{]} \\
exp & ::= & exp\texttt{(}exp\texttt{,}\ldots\texttt{)} \\
stmt & ::= & \texttt{return}\ exp \\
def & ::= & \texttt{def}\ var\texttt{(}var\texttt{:}type\texttt{,}\ldots\texttt{)}\ \texttt{->}\ type\texttt{:}\ stmt^{+} \\
\mathcal{L}_{\mathsf{Fun}} & ::= & def\ldots\ stmt\ldots
\end{array}
$$

Figure 7.1
The concrete syntax of $\mathcal{L}_{\mathsf{Fun}}$, extending $\mathcal{L}_{\mathsf{Tup}}$ (Figure 6.1).

that can be referenced from inside a function body are other globally-defined functions. The syntax of $\mathcal{L}_{\mathsf{Fun}}$ 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 $\mathcal{L}_{\mathsf{Fun}}$. 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 `(0, 41)`. The result is `(1, 42)`, from which we return the `42`.

The definitional interpreter for $\mathcal{L}_{\mathsf{Fun}}$ is in Figure 7.4. The case for the `Module` AST is responsible for setting up the mutual recursion between the top-level function definitions. We create a dictionary named `env` and fill it in by mapping each function name to a new `Function` value, each of which stores a reference to the `env`. (We define the class `Function` for this purpose.) To interpret a function call, we match the result of the function expression to obtain a function value. We then extend the function's environment with mapping of parameters to argument values. Finally, we interpret the body of the function in this extended environment.

The type checker for $\mathcal{L}_{\mathsf{Fun}}$ is in Figure 7.5. (We omit the code that parses function parameters into the simpler abstract syntax.) Similar to the interpreter, the case for the `Module` 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 call, 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 call as a whole is the return type from the function type.

$$
\begin{array}{rcl}
\mathit{binaryop} & ::= & \texttt{Add()} \mid \texttt{Sub()} \\
\mathit{unaryop} & ::= & \texttt{USub()} \\
\mathit{exp} & ::= & \texttt{Constant(}\mathit{int}\texttt{)} \mid \texttt{Call(Name('input\_int'),[])} \\
& \mid & \texttt{UnaryOp(}\mathit{unaryop}\texttt{,}\mathit{exp}\texttt{)} \mid \texttt{BinOp(}\mathit{binaryop}\texttt{,}\mathit{exp}\texttt{,}\mathit{exp}\texttt{)} \\
\mathit{stmt} & ::= & \texttt{Expr(Call(Name('print'),[}\mathit{exp}\texttt{]))} \mid \texttt{Expr(}\mathit{exp}\texttt{)} \\
\hline
\mathit{exp} & ::= & \texttt{Name(}\mathit{var}\texttt{)} \\
\mathit{stmt} & ::= & \texttt{Assign([Name(}\mathit{var}\texttt{)], }\mathit{exp}\texttt{)} \\
\hline
\mathit{boolop} & ::= & \texttt{And()} \mid \texttt{Or()} \\
\mathit{unaryop} & ::= & \texttt{Not()} \\
\mathit{cmp} & ::= & \texttt{Eq()} \mid \texttt{NotEq()} \mid \texttt{Lt()} \mid \texttt{LtE()} \mid \texttt{Gt()} \mid \texttt{GtE()} \\
\mathit{bool} & ::= & \texttt{True} \mid \texttt{False} \\
\mathit{exp} & ::= & \texttt{Constant(}\mathit{bool}\texttt{)} \mid \texttt{BoolOp(}\mathit{boolop}\texttt{,[}\mathit{exp}\texttt{,}\mathit{exp}\texttt{])} \\
& \mid & \texttt{Compare(}\mathit{exp}\texttt{,[}\mathit{cmp}\texttt{],[}\mathit{exp}\texttt{])} \mid \texttt{IfExp(}\mathit{exp}\texttt{,}\mathit{exp}\texttt{,}\mathit{exp}\texttt{)} \\
\mathit{stmt} & ::= & \texttt{If(}\mathit{exp}\texttt{, }\mathit{stmt}^{+}\texttt{, }\mathit{stmt}^{+}\texttt{)} \\
\hline
\mathit{stmt} & ::= & \texttt{While(}\mathit{exp}\texttt{, }\mathit{stmt}^{+}\texttt{, [])} \\
\hline
\mathit{cmp} & ::= & \texttt{Is()} \\
\mathit{exp} & ::= & \texttt{Tuple(}\mathit{exp}^{+}\texttt{,Load())} \mid \texttt{Subscript(}\mathit{exp}\texttt{,Constant(}\mathit{int}\texttt{),Load())} \\
& \mid & \texttt{Call(Name('len'),[}\mathit{exp}\texttt{])} \\
\hline
\mathit{type} & ::= & \texttt{IntType()} \mid \texttt{BoolType()VoidType()} \mid \texttt{TupleType[}\mathit{type}^{+}\texttt{]} \\
& \mid & \texttt{FunctionType(}\mathit{type}^{*}\texttt{, }\mathit{type}\texttt{)} \\
\mathit{exp} & ::= & \texttt{Call(}\mathit{exp}\texttt{, }\mathit{exp}^{*}\texttt{)} \\
\mathit{stmt} & ::= & \texttt{Return(}\mathit{exp}\texttt{)} \\
\mathit{params} & ::= & (\mathit{var}\texttt{,}\mathit{type})^{*} \\
\mathit{def} & ::= & \texttt{FunctionDef(}\mathit{var}\texttt{, }\mathit{params}\texttt{, }\mathit{stmt}^{+}\texttt{, None, }\mathit{type}\texttt{, None)} \\
\mathcal{L}_{\mathsf{Fun}} & ::= & \texttt{Module([}\mathit{def}\ldots\mathit{stmt}\ldots\texttt{])}
\end{array}
$$

Figure 7.2
The abstract syntax of $\mathcal{L}_{\mathsf{Fun}}$, extending $\mathcal{L}_{\mathsf{Tup}}$ (Figure 6.2).

```
def map(f : Callable[[int], int], v : tuple[int,int]) -> tuple[int,int]:
    return f(v[0]), f(v[1])

def inc(x : int) -> int:
    return x + 1

print( map(inc, (0, 41))[1] )
```

Figure 7.3
Example of using functions in $\mathcal{L}_{\mathsf{Fun}}$.

```
class InterpLfun(InterpLtup):

  def apply_fun(self, fun, args, e):
      match fun:
        case Function(name, xs, body, env):
          new_env = env.copy().update(zip(xs, args))
          return self.interp_stmts(body, new_env)
        case _:
          raise Exception('apply_fun: unexpected: ' + repr(fun))

  def interp_exp(self, e, env):
    match e:
      case Call(Name('input_int'), []):
        return super().interp_exp(e, env)
      case Call(func, args):
        f = self.interp_exp(func, env)
        vs = [self.interp_exp(arg, env) for arg in args]
        return self.apply_fun(f, vs, e)
      case _:
        return super().interp_exp(e, env)

  def interp_stmts(self, ss, env):
    if len(ss) == 0:
      return
    match ss[0]:
      case Return(value):
        return self.interp_exp(value, env)
      case FunctionDef(name, params, bod, dl, returns, comment):
        ps = [x for (x,t) in params]
        env[name] = Function(name, ps, bod, env)
        return self.interp_stmts(ss[1:], env)
      case _:
        return super().interp_stmts(ss, env)

  def interp(self, p):
    match p:
      case Module(ss):
        env = {}
        self.interp_stmts(ss, env)
        if 'main' in env.keys():
            self.apply_fun(env['main'], [], None)
      case _:
        raise Exception('interp: unexpected ' + repr(p))
```

Figure 7.4
Interpreter for the $\mathcal{L}_{\mathsf{Fun}}$ language.

```
class TypeCheckLfun(TypeCheckLtup):
  def type_check_exp(self, e, env):
    match e:
      case Call(Name('input_int'), []):
        return super().type_check_exp(e, env)
      case Call(func, args):
        func_t = self.type_check_exp(func, env)
        args_t = [self.type_check_exp(arg, env) for arg in args]
        match func_t:
          case FunctionType(params_t, return_t):
            for (arg_t, param_t) in zip(args_t, params_t):
                check_type_equal(param_t, arg_t, e)
            return return_t
          case _:
            raise Exception('type_check_exp: in call, unexpected ' +
                            repr(func_t))
      case _:
        return super().type_check_exp(e, env)

  def type_check_stmts(self, ss, env):
    if len(ss) == 0:
      return
    match ss[0]:
      case FunctionDef(name, params, body, dl, returns, comment):
        new_env = env.copy().update(params)
        rt = self.type_check_stmts(body, new_env)
        check_type_equal(returns, rt, ss[0])
        return self.type_check_stmts(ss[1:], env)
      case Return(value):
        return self.type_check_exp(value, env)
      case _:
        return super().type_check_stmts(ss, env)

  def type_check(self, p):
    match p:
      case Module(body):
        env = {}
        for s in body:
          match s:
            case FunctionDef(name, params, bod, dl, returns, comment):
              if name in env:
                raise Exception('type_check: function ' +
                                repr(name) + ' defined twice')
              params_t = [t for (x,t) in params]
              env[name] = FunctionType(params_t, returns)
        self.type_check_stmts(body, env)
      case _:
        raise Exception('type_check: unexpected ' + repr(p))
```

Figure 7.5

Type checker for the $\mathcal{L}_{\mathsf{Fun}}$ 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 and instruction-pointer relative addressing. For example, the following puts the address of the `inc` label into the `rbx` register.

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

Recall from Section 6.6 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 need to make an *indirect function call*. The x86 syntax for this is a `callq` instruction but with an asterisk before the register name.

```
callq *%rbx
```

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. Also recall that the register `rax` is for the return value of the function.

Regarding (2) frames and the procedure call stack, recall from Section 2.2 that the stack grows down 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.

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

Figure 7.6
Memory layout of caller and callee frames.

Recall that we are storing variables of tuple type on the root stack. So the prelude needs to move the root stack pointer `r15` up according to the number 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 pointer to a vector. Otherwise the garbage collector will interpret the garbage bits in those slots as memory addresses and try to traverse them, causing serious mayhem!

Regarding (3) the sharing of registers between different functions, recall from Section 3.1 that the registers are divided into two groups, the caller-saved registers and the callee-saved registers. The caller should assume that all the caller-saved registers get overwritten with arbitrary values by the callee. 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, …, f_n, then the amount of stack space is linear in n. The depth n can grow quite large if functions are (mutually) 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.

```
def tail_sum(n : int, r : int) -> int:
    if n == 0:
        return r
    else:
        return tail_sum(n - 1, n + r)

print( 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, which we describe in the 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 unnecessarily overwrite the return address. Instead we can simply use the `jmp` instruction. Like the indirect function call, we write an *indirect jump* with a register prefixed with an asterisk. We recommend using `rax` to hold the jump target because the preceding conclusion can overwrite just about everything else.

$$\begin{array}{lcl} \mathit{exp} & ::= & \texttt{FunRef(}\mathit{var}\texttt{,}\ \mathit{int}\texttt{)} \\ \mathcal{L}_{\mathsf{FunRef}} & ::= & \texttt{Module([}\mathit{def}\texttt{,}\ \ldots\texttt{])} \end{array}$$

Figure 7.7
The abstract syntax $\mathcal{L}_{\mathsf{FunRef}}$, an extension of $\mathcal{L}_{\mathsf{Fun}}$ (Figure 7.2).

```
jmp *%rax
```

7.3 Shrink $\mathcal{L}_{\mathsf{Fun}}$

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.

`Module(`*def* … *stmt* …`)`
$\Rightarrow$ `Module(`*def* … *mainDef*`)`

where *mainDef* is

`FunctionDef('main', [], int, None, `*stmt* …` Return(Constant(0)), None)`

7.4 Reveal Functions and the $\mathcal{L}_{\mathsf{FunRef}}$ language

The syntax of $\mathcal{L}_{\mathsf{Fun}}$ 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; we need to 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 `Name(`f`)` to `FunRef(`f`, `n`)` where n is the arity of the function.[8] This pass is named `reveal_functions` and the output language, $\mathcal{L}_{\mathsf{FunRef}}$, is defined in Figure 7.7.

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 $\mathcal{L}_{\mathsf{Fun}}$ 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 vector and passes it as the sixth argument.

8. The arity is not needed in this chapter but is used in Chapter 9.

Each function definition with seven or more parameters is transformed as follows.

FunctionDef(f, [(x_1,T_1), ..., (x_n,T_n)], T_r, None, *body*, None)
$\Rightarrow$
FunctionDef(f, [(x_1,T_1), ..., (x_5,T_5),(tup,TupleType([T_6, ..., T_n]))],
T_r, None, *body*$'$, None)

where the *body* is transformed into *body*$'$ by replacing the occurrences of each parameter x_i where $i > 5$ with the kth element of the tuple, where $k = i - 6$.

Name(x_i) $\Rightarrow$ Subscript(tup, Constant(k), Load())

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

Call(e_0, [e_1, ..., e_n]) $\Rightarrow$ Call(e_0, [e_1, ..., e_5,Tuple([e_6, ..., e_n])])

7.6 Remove Complex Operands

The primary decisions to make for this pass is whether to classify `FunRef` and `Call` as either atomic or complex expressions. Recall that a simple expression will eventually end up as just an immediate argument of an x86 instruction. Function application will be translated to a sequence of instructions, so `Call` must be classified as complex expression. On the other hand, the arguments of `Call` 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, $\mathcal{L}^{mon}_{\mathsf{FunRef}}$, extends $\mathcal{L}^{mon}_{\mathsf{Alloc}}$ (Figure 6.11) with `FunRef` and `Call` in the grammar for expressions. Also, $\mathcal{L}^{mon}_{\mathsf{FunRef}}$ adds `Return` to the grammar for statements.

7.7 Explicate Control and the $\mathcal{C}_{\mathsf{Fun}}$ language

Figure 7.8 defines the abstract syntax for $\mathcal{C}_{\mathsf{Fun}}$, the output of `explicate_control`. The auxiliary functions for assignment should be updated with cases for `Call` and `FunRef` and the function for predicate context should be updated for `Call` but not `FunRef`. (A `FunRef` cannot be a Boolean.) In assignment and predicate contexts, `Apply` becomes `Call`. We recommend defining a new auxiliary function for processing function definitions. This code is similar to the case for `Program` in $\mathcal{L}_{\mathsf{Tup}}$. The top-level `explicate_control` function that handles the `ProgramDefs` form of $\mathcal{L}_{\mathsf{Fun}}$ can then apply this new function to all the function definitions.

The translation of `Return` statements requires a new auxiliary function to handle expressions in tail context, called `explicate_tail`. The function should take an expression and the dictionary of basic blocks and produce a list of statements in the $\mathcal{C}_{\mathsf{Fun}}$ language. The `explicate_tail` function should include cases for `Begin`, `IfExp`, `Let`, `Call`, and a default case for other kinds of expressions. The default case should

```
atm    ::= Constant(int) | Name(var) | Constant(bool)
exp    ::= atm | Call(Name('input_int'),[])
         | BinOp(atm,binaryop,atm) | UnaryOp(unaryop,atm)
         | Compare(atm,[cmp],[atm])
stmt   ::= Expr(Call(Name('print'),[atm])) | Expr(exp)
         | Assign([Name(var)], exp) | Return(exp) | Goto(label)
         | If(Compare(atm,[cmp],[atm]), [Goto(label)], [Goto(label)])
---------------------------------------------------------------------
exp    ::= Subscript(atm,atm,Load()) | Allocate(int,type)
         | GlobalValue(var) | Call(Name('len'),[atm])
stmt   ::= Collect(int)
         | Assign([Subscript(atm,atm,Store())], atm)
---------------------------------------------------------------------
exp     ::= FunRef(label, int) | Call(atm, atm*)
stmt    ::= TailCall(atm, atm*)
params  ::= [(var,type),...]
block   ::= label:stmt*
blocks  ::= {block,...}
def     ::= FunctionDef(label, params, blocks, None, type, None)
C_Fun   ::= CProgramDefs([def,...])
```

Figure 7.8
The abstract syntax of $\mathcal{C}_{\mathsf{Fun}}$, extending $\mathcal{C}_{\mathsf{Tup}}$ (Figure 6.12).

```
arg         ::= $int | %reg | int(%reg) | %bytereg | var(%rip)
cc          ::= e | ne | l | le | g | ge
instr       ::= ... | callq *arg | tailjmp arg | leaq arg,%reg
block       ::= instr+
def         ::= .globl label (label: block)*
x86_callq*  ::= def ...
```

Figure 7.9
The concrete syntax of $\text{x86}_{\mathsf{callq*}}$ (extends $\text{x86}_{\mathsf{Global}}$ of Figure 6.13).

produce a `Return` statement. The case for `Call` should change it into `TailCall`. The other cases should recursively process their subexpressions and statements, choosing the appropriate explicate functions for the various contexts.

7.8 Select Instructions and the $\text{x86}_{\mathsf{callq*}}$ Language

The output of select instructions is a program in the $\text{x86}_{\mathsf{callq*}}$ language, whose syntax is defined in Figure 7.10.

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 $\mathcal{C}_{\mathsf{Fun}}$ to *arg* in $\text{x86}^{\mathsf{Var}}_{\mathsf{callq*}}$.

```
lhs = FunRef(f, n);        ⇒        leaq f(%rip), lhs′
```

$$
\begin{array}{lcl}
\mathit{arg} & ::= & \texttt{Constant(}\mathit{int}\texttt{)} \mid \texttt{Reg(}\mathit{reg}\texttt{)} \mid \texttt{Deref(}\mathit{reg}\texttt{,}\mathit{int}\texttt{)} \mid \texttt{ByteReg(}\mathit{reg}\texttt{)} \\
 & \mid & \texttt{Global(}\mathit{var}\texttt{)} \mid \texttt{FunRef(}\mathit{label}\texttt{, }\mathit{int}\texttt{)} \\
\mathit{instr} & ::= & \ldots \mid \texttt{IndirectCallq(}\mathit{arg}\texttt{, }\mathit{int}\texttt{)} \mid \texttt{TailJmp(}\mathit{arg}\texttt{, }\mathit{int}\texttt{)} \\
 & \mid & \texttt{Instr('leaq',[}\mathit{arg}\texttt{,Reg(}\mathit{reg}\texttt{)])} \\
\mathit{block} & ::= & \mathit{label}\texttt{:}\ \mathit{instr}^* \\
\mathit{blocks} & ::= & \texttt{\{}\mathit{block}\texttt{, }\ldots\texttt{\}} \\
\mathit{def} & ::= & \texttt{FunctionDef(}\mathit{label}\texttt{, [], }\mathit{blocks}\texttt{, \_, }\mathit{type}\texttt{, \_)} \\
\text{x86}_{\mathsf{callq*}} & ::= & \texttt{X86ProgramDefs([}\mathit{def}\texttt{, }\ldots\texttt{])}
\end{array}
$$

Figure 7.10
The abstract syntax of $\text{x86}_{\mathsf{callq*}}$ (extends $\text{x86}_{\mathsf{Global}}$ of Figure 6.14).

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 to these local variables.

$$
\begin{array}{l}
\texttt{FunctionDef(}f\texttt{, [(}x_1\texttt{,}T_1\texttt{), }\ldots\texttt{], }B\texttt{, \_, }T_r\texttt{, \_)} \\
\Rightarrow \\
\texttt{FunctionDef(}f\texttt{, [], }B'\texttt{, \_, int, \_)}
\end{array}
$$

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 variables. So the `start` block of B shown on the left is changed to the code on the right.

$$
\begin{array}{l}
\texttt{start:} \\
\quad \mathit{instr}_1 \\
\quad \cdots \\
\quad \mathit{instr}_n
\end{array}
\quad\Rightarrow\quad
\begin{array}{l}
\texttt{start:} \\
\quad \texttt{movq \%rdi, } x_1 \\
\quad \cdots \\
\quad \mathit{instr}_1 \\
\quad \cdots \\
\quad \mathit{instr}_n
\end{array}
$$

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

lhs `= Call(`fun`,` $arg_1 \ldots$`)`
$\Rightarrow$
`movq` arg_1`, %rdi`
`movq` arg_2`, %rsi`
$\vdots$
`callq *`fun
`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 `TailJmp`. 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 2.5 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

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 `TailJmp` and `IndirectCallq` determines how many of the argument-passing registers should be considered as read by those instructions. Also, the target field of `TailJmp` 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 6.7 we discussed the need to spill vector-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 vector-typed variable when it is live during a call to `collect`, but we should spill the variable if it is live during call to a user-defined function. Thus, in the `build_interference` pass, we recommend adding interference edges between call-live vector-typed variables and the callee-saved registers (in addition to the usual addition of edges between call-live variables and the caller-saved registers).

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 7.10) 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 `TailJmp` is *rax*, our reserved register—mostly to make code generation more convenient, because we trample many 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 `TailJmp` into a sequence of instructions. A straightforward translation of `TailJmp` would simply be `jmp *`*arg*. However, before the jump we need to pop the current frame. 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, you need to 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 by the size of the stack frame for this function, which depends on the number of 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 vectors.
5. Initialize to zero all new entries in the root-stack frame.

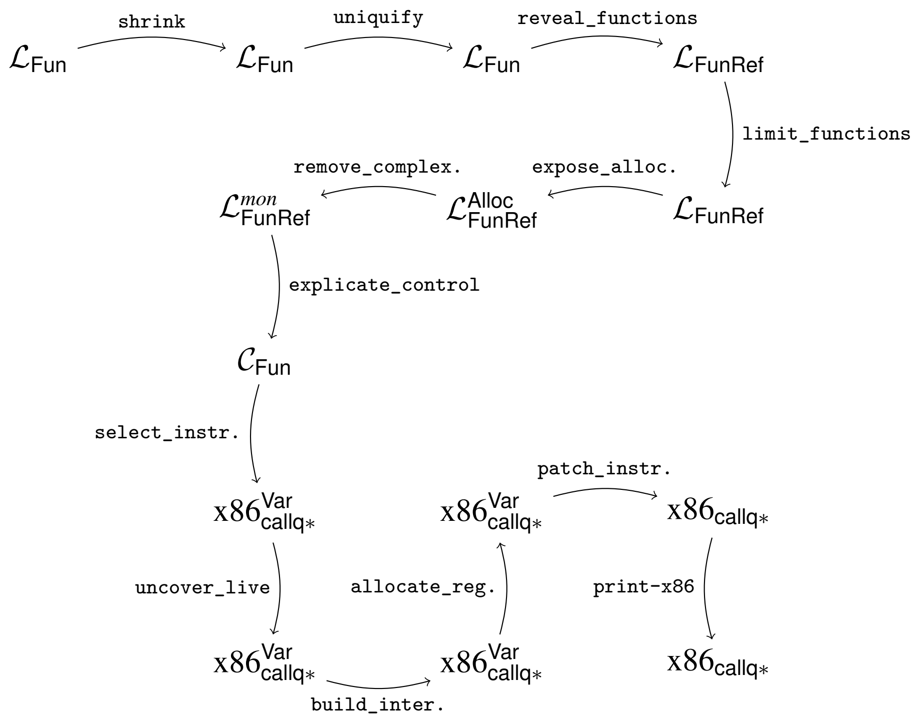


Figure 7.11
Diagram of the passes for $\mathcal{L}_{\mathsf{Fun}}$, a language with functions.

6. Jump to the start block.

The prelude of the `main` function has one additional task: call the `initialize` function to set up the garbage collector and 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 by the size of the stack frame for this function.
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 27 Expand your compiler to handle $\mathcal{L}_{\mathsf{Fun}}$ as outlined in this chapter. Create 5 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 7.11 gives an overview of the passes for compiling $\mathcal{L}_{\mathsf{Fun}}$ to x86.

7.12 An Example Translation

Figure 7.12 shows an example translation of a simple function in $\mathcal{L}_{\mathsf{Fun}}$ to x86. The figure also includes the results of the `explicate_control` and `select_instructions` passes.

```
def add(x:int, y:int) -> int:
    return x + y

print(add(40, 2))
```

⇓

```
def add(x:int, y:int) -> int:
    addstart:
        return x + y

def main() -> int:
    mainstart:
        fun.0 = add
        tmp.1 = fun.0(40, 2)
        print(tmp.1)
        return 0
```

⇒

```
def add() -> int:
    addstart:
        movq %rdi, x
        movq %rsi, y
        movq x, %rax
        addq y, %rax
        jmp addconclusion

def main() -> int:
    mainstart:
        leaq add, fun.0
        movq $40, %rdi
        movq $2, %rsi
        callq *fun.0
        movq %rax, tmp.1
        movq tmp.1, %rdi
        callq print_int
        movq $0, %rax
        jmp mainconclusion
```

⇓

```
  .align 16
add:
  pushq %rbp
  movq %rsp, %rbp
  subq $0, %rsp
  jmp addstart
addstart:
  movq %rdi, %rdx
  movq %rsi, %rcx
  movq %rdx, %rax
  addq %rcx, %rax
  jmp addconclusion
addconclusion:
  subq $0, %r15
  addq $0, %rsp
  popq %rbp
  retq
```

```
  .globl main
  .align 16
main:
  pushq %rbp
  movq %rsp, %rbp
  subq $0, %rsp
  movq $65536, %rdi
  movq $65536, %rsi
  callq initialize
  movq rootstack_begin(%rip), %r15
  jmp mainstart
mainstart:
  leaq add(%rip), %rcx
  movq $40, %rdi
  movq $2, %rsi
  callq *%rcx
  movq %rax, %rcx
  movq %rcx, %rdi
  callq print_int
  movq $0, %rax
  jmp mainconclusion
mainconclusion:
  subq $0, %r15
  addq $0, %rsp
  popq %rbp
  retq
```

Figure 7.12
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 $\mathcal{L}_\lambda$, which extends $\mathcal{L}_{\mathsf{Fun}}$ with lexically scoped functions using the `lambda` form. The body of the `lambda` refers to three variables: `x`, `y`, and `z`. The binding sites for `x` and `y` are outside of the `lambda`. Variable `y` is a local variable of function `f` and `x` is a parameter of function `f`. The `lambda` is returned from the function `f`. The main expression of the program includes two calls to `f` with different arguments for `x`, first `5` then `3`. The functions returned from `f` are bound to variables `g` and `h`. Even though these two functions were created by the same `lambda`, they are really different functions because they use different values for `x`. Applying `g` to `11` produces `20` whereas applying `h` to `15` produces `22`. The result of this program is `42`.

The approach that we take for implementing lexically scoped functions is to compile them into top-level function definitions, translating from $\mathcal{L}_\lambda$ into $\mathcal{L}_{\mathsf{Fun}}$. 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 $\mathcal{L}_{\mathsf{Fun}}$ function may not refer to variables defined outside of it. To identify such variable occurrences, we review the standard notion of free variable.

Definition 1 *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.*

```
def f(x : int) -> Callable[[int], int]:
    y = 4
    return lambda z: x + y + z

g = f(5)
h = f(3)
print( g(11) + h(15) )
```

Figure 8.1
Example of a lexically scoped function.

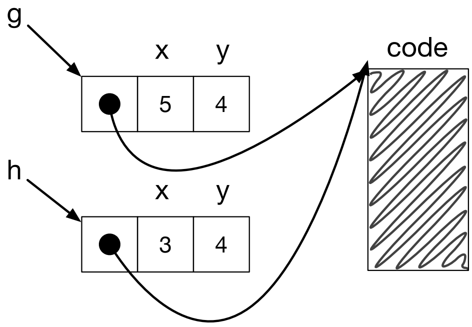


Figure 8.2
Flat closure representations for the two functions produced by the `lambda` in Figure 8.1.

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: 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"). Fortunately, 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's 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 in the first element of the closure and call it, passing in the closure itself and then the regular arguments, in this case `11`. This technique for applying a closure 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 free variable, to bind those variables to the appropriate elements from the closure

exp	::=	*int* | `input_int()` | `-`*exp* | *exp* `+` *exp* | *exp* `-` *exp* | `(`*exp*`)`
stmt	::=	`print(`*exp*`)` | *exp*
exp	::=	*var*
stmt	::=	*var* `=` *exp*
cmp	::=	`==` | `!=` | `<` | `<=` | `>` | `>=`
exp	::=	`True` | `False` | *exp* `and` *exp* | *exp* `or` *exp* | `not` *exp*
	|	*exp* *cmp* *exp* | *exp* `if` *exp* `else` *exp*
stmt	::=	`if` *exp*`:` $stmt^+$ `else:` $stmt^+$
stmt	::=	`while` *exp*`:` $stmt^+$
cmp	::=	`is`
exp	::=	*exp*`,` … `,`*exp* | *exp*`[`*int*`]` | `len(`*exp*`)`
type	::=	`int` | `bool` | `tuple[`$type^+$`]` | `Callable[[`*type*`,` …`],` *type*`]`
exp	::=	*exp*`(`*exp*`,` …`)`
stmt	::=	`return` *exp*
def	::=	`def` *var*`(`*var*`:`*type*`,` …`) ->` *type*`:` $stmt^+$
exp	::=	`lambda` *var*`,` … `:` *exp* | `arity(`*exp*`)`
stmt	::=	*var* `:` *type* `=` *exp*
$\mathcal{L}_{\mathsf{Fun}}$	::=	*def* … *stmt* …

Figure 8.3
The concrete syntax of $\mathcal{L}_\lambda$, extending $\mathcal{L}_{\mathsf{Fun}}$ (Figure 7.1) with `lambda`.

parameter. This three-step dance is known as *closure conversion*. We discuss the details of closure conversion in Section 8.5 and the code generated from the example in Section 8.6. But first we define the syntax and semantics of $\mathcal{L}_\lambda$ in Section 8.1.

8.1 The $\mathcal{L}_\lambda$ Language

The concrete and abstract syntax for $\mathcal{L}_\lambda$, 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 $\mathcal{L}_{\mathsf{Fun}}$, which already has syntax for function application. The syntax also includes an assignment statement that includes a type annotation for the variable on the left-hand side, which facilitates the type checking of `lambda` expressions that we discuss later in this section. The `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. The `arity` operation is not in Python, but the same functionality is available in a more complex form. We include `arity` in the $\mathcal{L}_\lambda$ source language to enable testing.

Figure 8.5 shows the definitional interpreter for $\mathcal{L}_\lambda$. 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.

Figures 8.6 and 8.7 define the type checker for $\mathcal{L}_\lambda$, which is more complex than one might expect. The reason for the added complexity is that the syntax of `lambda` does not include type annotations for the parameters or return type. Instead they must

```
binaryop ::= Add() | Sub()
unaryop  ::= USub()
exp      ::= Constant(int) | Call(Name('input_int'),[])
          |  UnaryOp(unaryop,exp) | BinOp(binaryop,exp,exp)
stmt     ::= Expr(Call(Name('print'),[exp])) | Expr(exp)
-----------------------------------------------------------------
exp  ::= Name(var)
stmt ::= Assign([Name(var)], exp)
-----------------------------------------------------------------
boolop  ::= And() | Or()
unaryop ::= Not()
cmp     ::= Eq() | NotEq() | Lt() | LtE() | Gt() | GtE()
bool    ::= True | False
exp     ::= Constant(bool) | BoolOp(boolop,[exp,exp])
         |  Compare(exp,[cmp],[exp]) | IfExp(exp,exp,exp)
stmt    ::= If(exp, stmt+, stmt+)
-----------------------------------------------------------------
stmt ::= While(exp, stmt+, [])
-----------------------------------------------------------------
cmp ::= Is()
exp ::= Tuple(exp+,Load()) | Subscript(exp,Constant(int),Load())
     |  Call(Name('len'),[exp])
-----------------------------------------------------------------
type   ::= IntType() | BoolType()VoidType() | TupleType[type+]
        |  FunctionType(type*, type)
exp    ::= Call(exp, exp*)
stmt   ::= Return(exp)
params ::= (var,type)*
def    ::= FunctionDef(var, params, stmt+, None, type, None)
-----------------------------------------------------------------
exp  ::= Lambda(var*, exp) | Call(Name('arity'), [exp])
stmt ::= AnnAssign(var, type, exp, 0)
Lλ   ::= Module([def ... stmt ...])
```

Figure 8.4
The abstract syntax of $\mathcal{L}_\lambda$, extending $\mathcal{L}_{\mathsf{Fun}}$ (Figure 7.2).

be inferred. There are many approaches of type inference to choose from of varying degrees of complexity. We choose one of the simpler approaches, bidirectional type inference (Dunfield and Krishnaswami 2021) (aka. local type inference (Pierce and Turner 2000)), because the focus of this book is compilation, not type inference.

The main idea of bidirectional type inference is to add an auxilliary function, here named `check_exp`, that takes an expected type and checks whether the given expression is of that type. Thus, in `check_exp`, type information flows in a top-down manner with respect to the AST, in contrast to the regular `type_check_exp` function, where type information flows in a primarily bottom-up manner. The idea then is to use `check_exp` in all the places where we already know what the type of an expression should be, such as in the `return` statement of a top-level function definition, or on the right-hand side of an annotated assignment statement.

Getting back to `lambda`, it is straightforward to check a `lambda` inside `check_exp` because the expected type provides the parameter types and the return type. On the other hand, inside `type_check_exp` we disallow `lambda`, which means that we do

```
class InterpLlambda(InterpLfun):

  def arity(self, v):
    match v:
      case Function(name, params, body, env):
        return len(params)
      case _:
        raise Exception('Llambda arity unexpected ' + repr(v))

  def interp_exp(self, e, env):
    match e:
      case Call(Name('arity'), [fun]):
        f = self.interp_exp(fun, env)
        return self.arity(f)
      case Lambda(params, body):
        return Function('lambda', params, [Return(body)], env)
      case _:
        return super().interp_exp(e, env)

  def interp_stmts(self, ss, env):
    if len(ss) == 0:
      return
    match ss[0]:
      case AnnAssign(lhs, typ, value, simple):
        env[lhs.id] = self.interp_exp(value, env)
        return self.interp_stmts(ss[1:], env)
      case _:
        return super().interp_stmts(ss, env)
```

Figure 8.5
Interpreter for $\mathcal{L}_\lambda$.

not allow `lambda` in contexts where we don't already know its type. This restriction does not incur a loss of expressiveness for $\mathcal{L}_\lambda$ because it is straightforward to modify a program to sidestep the restriction, for example, by using an annotated assignment statement to assign the `lambda` to a temporary variable.

Note that for the `Name` and `Lambda` AST nodes, the type checker records their type in a `has_type` field. This type information is used later in this chapter.

```
class TypeCheckLlambda(TypeCheckLfun):

  def type_check_exp(self, e, env):
    match e:
      case Name(id):
        e.has_type = env[id]
        return env[id]
      case Lambda(params, body):
        raise Exception('cannot synthesize a type for a lambda')
      case Call(Name('arity'), [func]):
        func_t = self.type_check_exp(func, env)
        match func_t:
          case FunctionType(params_t, return_t):
            return IntType()
          case _:
            raise Exception('in arity, unexpected ' + repr(func_t))
      case _:
        return super().type_check_exp(e, env)

  def check_exp(self, e, ty, env):
    match e:
      case Lambda(params, body):
        e.has_type = ty
        match ty:
          case FunctionType(params_t, return_t):
            new_env = env.copy().update(zip(params, params_t))
            self.check_exp(body, return_t, new_env)
          case _:
            raise Exception('lambda does not have type ' + str(ty))
      case Call(func, args):
        func_t = self.type_check_exp(func, env)
        match func_t:
          case FunctionType(params_t, return_t):
            for (arg, param_t) in zip(args, params_t):
                self.check_exp(arg, param_t, env)
            self.check_type_equal(return_t, ty, e)
          case _:
            raise Exception('type_check_exp: in call, unexpected ' + \
                            repr(func_t))
      case _:
        t = self.type_check_exp(e, env)
        self.check_type_equal(t, ty, e)
```

Figure 8.6
Type checking $\mathcal{L}_\lambda$, part 1.

```
def check_stmts(self, ss, return_ty, env):
  if len(ss) == 0:
    return
  match ss[0]:
    case FunctionDef(name, params, body, dl, returns, comment):
      new_env = env.copy().update(params)
      rt = self.check_stmts(body, returns, new_env)
      self.check_stmts(ss[1:], return_ty, env)
    case Return(value):
      self.check_exp(value, return_ty, env)
    case Assign([Name(id)], value):
      if id in env:
        self.check_exp(value, env[id], env)
      else:
        env[id] = self.type_check_exp(value, env)
      self.check_stmts(ss[1:], return_ty, env)
    case Assign([Subscript(tup, Constant(index), Store())], value):
      tup_t = self.type_check_exp(tup, env)
      match tup_t:
        case TupleType(ts):
          self.check_exp(value, ts[index], env)
        case _:
          raise Exception('expected a tuple, not ' + repr(tup_t))
      self.check_stmts(ss[1:], return_ty, env)
    case AnnAssign(Name(id), ty_annot, value, simple):
      ss[0].annotation = ty_annot
      if id in env:
          self.check_type_equal(env[id], ty_annot)
      else:
          env[id] = ty_annot
      self.check_exp(value, ty_annot, env)
      self.check_stmts(ss[1:], return_ty, env)
    case _:
      self.type_check_stmts(ss, env)

def type_check(self, p):
  match p:
    case Module(body):
      env = {}
      for s in body:
          match s:
            case FunctionDef(name, params, bod, dl, returns, comment):
              params_t = [t for (x,t) in params]
              env[name] = FunctionType(params_t, returns)
      self.check_stmts(body, int, env)
```

Figure 8.7
Type checking the `lambda`'s in $\mathcal{L}_\lambda$, part 2.

8.2 Assignment and Lexically Scoped Functions

The combination of lexically-scoped functions and assignment to variables raises a challenge with our 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`.

```
def g(z : int) -> int:
    x = 0
    y = 0
    f : Callable[[int],int] = lambda a: a + x + z
    x = 10
    y = 12
    return f(y)

print( g(20) )
```

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 8.5) generates code for the `lambda` that copies the old value of `x` into a closure. Thus, if we naively add support for assignment to our current compiler, 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.

```
def f():
    x = 0
    g = lambda: x
    x = 42
    return g

print( 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 shall introduce a new pass named `convert_assignments` in Section 8.4 to address this challenge.

8.3 Uniquify Variables

With the addition of `lambda` we have a complication to deal with: name shadowing. Consider the following program with a function `f` that has a parameter `x`. Inside `f` there are two `lambda` expressions. The first `lambda` has a parameter that is also named `x`.

```
def f(x:int, y:int) -> Callable[[int], int]:
  g : Callable[[int],int] = (lambda x: x + y)
  h : Callable[[int],int] = (lambda y: x + y)
  x = input_int()
  return g

print(f(0, 10)(32))
```

Many of our compiler passes rely on being able to connect variable uses with their definitions using just the name of the variable, including new passes in this chapter. However, in the above example the name of the variable does not uniquely determine its definition. To solve this problem we recommend implementing a pass named `uniquify` that renames every variable in the program to make sure they are all unique.

The following shows the result of `uniquify` for the above example. The `x` parameter of `f` is renamed to `x_0` and the `x` parameter of the `lambda` is renamed to `x_4`.

```
def f(x_0:int, y_1:int) -> Callable[[int], int] :
  g_2 : Callable[[int], int] = (lambda x_4: x_4 + y_1)
  h_3 : Callable[[int], int] = (lambda y_5: x_0 + y_5)
  x_0 = input_int()
  return g_2

def main() -> int :
  print(f(0, 10)(32))
  return 0
```

8.4 Assignment Conversion

The purpose of the `convert_assignments` pass is to address the challenge posed in Section 8.2 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 quite conservative and it is possible to develop a more liberal condition.

Consider again the first example from Section 8.2:

```
def g(z : int) -> int:
    x = 0
    y = 0
    f : Callable[[int],int] = lambda a: a + x + z
    x = 10
    y = 12
    return f(y)

print( g(20) )
```

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.

```
def g(z : int)-> int:
  x = (uninitialized(int),)
  x[0] = 0
  y = 0
  f : Callable[[int], int] = (lambda a: a + x[0] + z)
  x[0] = 10
  y = 12
  return f(y)

def main() -> int:
  print(g(20))
  return 0
```

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 defining an auxiliary function named `assigned_vars_stmt` that returns the set of variables that occur in the left-hand side of an assignment statement, and otherwise returns the empty set.

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 `Name(`x`)`, if x is in *AF*, then unbox it by translating `Name(`x`)` to a tuple read.

```
Name(x)
⇒
Subscript(Name(x), Constant(0), Load())
```

In the case for assignment, recursively process the right-hand side *rhs* to obtain *rhs′*. If *x* is in *AF*, translate the assignment into a tuple-write as follows.

```
Assign([Name(x)],rhs)
⇒
Assign([Subscript(Name(x), Constant(0), Store())], rhs′)
```

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.

```
def g(x : int) -> int:
  f : Callable[[int],int] = lambda a: a + x
  x = 10
  return 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 we has been renamed.

```
def g(x_0 : int)-> int:
  x = (x_0,)
  f : Callable[[int], int] = (lambda a: a + x[0])
  x[0] = 10
  return f(32)
```

8.5 Closure Conversion

The compiling of lexically-scoped functions into top-level function definitions is accomplished in the pass `convert_to_closures` that comes after `reveal_functions` and before `limit_functions`.

As usual, we 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.

```
Lambda([x1, ...,xn], body)
⇒
Closure(n, [FunRef(name, n), 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 : closTy, ps′, ...) -> rt′:
  fvs₁ = clos[1]
  ...
  fvsₙ = clos[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 `Bottom()` and the rest of the element types are the types of the free variables in the lambda. We use `Bottom()` because it is non-trivial to give a type to the function in the closure's type.[9] 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.

```
FunctionType([T₁, ..., Tₙ], Tᵣ)
⇒
TupleType([FunctionType([TupleType([]), 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 $T'_1, \ldots, T'_n$. 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.

```
Call(e, [e₁, ..., eₙ])
⇒
Begin([Assign([tmp], e′)],
      Call(Subscript(Name(tmp), Constant(0)),
           [tmp, e′₁, ..., e′ₙ]))
```

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.

```
FunRef(f, n)        ⇒   Closure(n, [FunRef(f n)])
```

9. To give an accurate type to a closure, we would need to add existential types to the type checker (Minamide, Morrisett, and Harper 1996).

```
def f(x : int) -> Callable[[int], int]:
  y = 4
  return lambda z: x + y + z

g = f(5)
h = f(3)
print( g(11) + h(15) )
```

⇒

```
def lambda_0(fvs_1:tuple[bot,int,tuple[int]],z:int) -> int:
  x = fvs_1[1]
  y = fvs_1[2]
  return x + y[0] + z

def f(fvs_2:bot, x:int) -> tuple[Callable[[tuple[],int], int]]
  y = (777,)
  y[0] = 4
  return (lambda_0, x, y)

def main() -> int:
  g = (let clos_3 = (f,) in clos_3[0](clos_3, 5))
  h = (let clos_4 = (f,) in clos_4[0](clos_4, 3))
  print((let clos_5 = g in clos_5[0](clos_5, 11))
        + (let clos_6 = h in clos_6[0](clos_6, 15)))
  return 0
```

Figure 8.8
Example of closure conversion.

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.

8.6 An Example Translation

Figure 8.8 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 28 Expand your compiler to handle $\mathcal{L}_\lambda$ 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.

$$
\begin{array}{lcl}
\textit{atm} & ::= & \texttt{Constant(}\textit{int}\texttt{)} \mid \texttt{Name(}\textit{var}\texttt{)} \mid \texttt{Constant(}\textit{bool}\texttt{)} \\
\textit{exp} & ::= & \textit{atm} \mid \texttt{Call(Name('input\_int'),[])} \\
& \mid & \texttt{BinOp(}\textit{atm}\texttt{,}\textit{binaryop}\texttt{,}\textit{atm}\texttt{)} \mid \texttt{UnaryOp(}\textit{unaryop}\texttt{,}\textit{atm}\texttt{)} \\
& \mid & \texttt{Compare(}\textit{atm}\texttt{,[}\textit{cmp}\texttt{],[}\textit{atm}\texttt{])} \\
\textit{stmt} & ::= & \texttt{Expr(Call(Name('print'),[}\textit{atm}\texttt{]))} \mid \texttt{Expr(}\textit{exp}\texttt{)} \\
& \mid & \texttt{Assign([Name(}\textit{var}\texttt{)], }\textit{exp}\texttt{)} \mid \texttt{Return(}\textit{exp}\texttt{)} \mid \texttt{Goto(}\textit{label}\texttt{)} \\
& \mid & \texttt{If(Compare(}\textit{atm}\texttt{,[}\textit{cmp}\texttt{],[}\textit{atm}\texttt{]), [Goto(}\textit{label}\texttt{)], [Goto(}\textit{label}\texttt{)])} \\
\hline
\textit{exp} & ::= & \texttt{Subscript(}\textit{atm}\texttt{,}\textit{atm}\texttt{,Load())} \mid \texttt{Allocate(}\textit{int}\texttt{,}\textit{type}\texttt{)} \\
& \mid & \texttt{GlobalValue(}\textit{var}\texttt{)} \mid \texttt{Call(Name('len'),[}\textit{atm}\texttt{])} \\
\textit{stmt} & ::= & \texttt{Collect(}\textit{int}\texttt{)} \\
& \mid & \texttt{Assign([Subscript(}\textit{atm}\texttt{,}\textit{atm}\texttt{,Store())], }\textit{atm}\texttt{)} \\
\hline
\textit{exp} & ::= & \texttt{FunRef(}\textit{label}\texttt{, }\textit{int}\texttt{)} \mid \texttt{Call(}\textit{atm}\texttt{, }\textit{atm}^*\texttt{)} \\
\textit{stmt} & ::= & \texttt{TailCall(}\textit{atm}\texttt{, }\textit{atm}^*\texttt{)} \\
\textit{params} & ::= & \texttt{[(}\textit{var}\texttt{,}\textit{type}\texttt{), ...]} \\
\textit{block} & ::= & \textit{label}\texttt{:}\textit{stmt}^* \\
\textit{blocks} & ::= & \texttt{\{}\textit{block}\texttt{, ...\}} \\
\textit{def} & ::= & \texttt{FunctionDef(}\textit{label}\texttt{, }\textit{params}\texttt{, }\textit{blocks}\texttt{, None, }\textit{type}\texttt{, None)} \\
\hline
\textit{exp} & ::= & \texttt{Uninitialized(}\textit{type}\texttt{)} \mid \texttt{AllocateClosure(}\textit{len}\texttt{,}\textit{type}\texttt{,}\textit{arity}\texttt{)} \\
& \mid & \texttt{Call(Name('arity'), [}\textit{atm}\texttt{])} \\
\mathcal{C}_{\mathsf{Clos}} & ::= & \texttt{CProgramDefs([}\textit{def}\texttt{, ...])}
\end{array}
$$

Figure 8.9
The abstract syntax of $\mathcal{C}_{\mathsf{Clos}}$, extending $\mathcal{C}_{\mathsf{Fun}}$ (Figure 7.8).

8.7 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 6.3. The only difference is replacing the use of `Allocate(`*len*`, `*type*`)` with `AllocateClosure(`*len*`, `*type*`, `*arity*`)`.

8.8 Explicate Control and $\mathcal{C}_{\mathsf{Clos}}$

The output language of `explicate_control` is $\mathcal{C}_{\mathsf{Clos}}$ whose abstract syntax is defined in Figure 8.9. The differences with respect to $\mathcal{C}_{\mathsf{Fun}}$ are the additions of `Uninitialized`, `AllocateClosure`, and `arity` to the grammar for *exp*. The handling of them in the `explicate_control` pass is similar to the handling of other expressions such as primitive operators.

8.9 Select Instructions

Compile `AllocateClosure(`*len*`, `*type*`, `*arity*`)` in almost the same way as the `Allocate(`*len*`, `*type*`)` form (Section 6.6). 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 6.6 a portion of the 64-bit tag was not used. We store the arity in the 5 bits starting at position 58.

Compile a call to the `arity` operator to a sequence of instructions that access the tag from position 0 of the tuple (representing a closure) and extract the 5-bits starting at position 58 from the tag.

Figure 8.10 provides an overview of all the passes needed for the compilation of $\mathcal{L}_\lambda$.

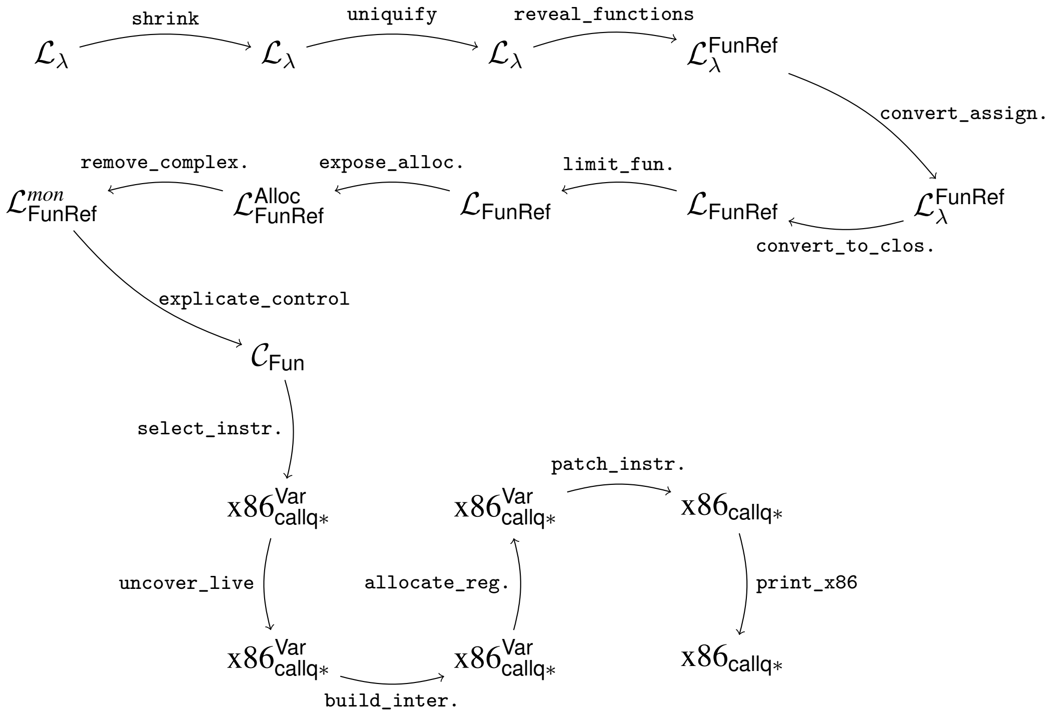


Figure 8.10
Diagram of the passes for $\mathcal{L}_\lambda$, a language with lexically-scoped functions.

8.10 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).

```
def tail_sum(n : int, s : int) -> int:
    if n == 0:
        return s
    else:
        return tail_sum(n - 1, n + s)

print( 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.

```
def tail_sum(fvs_3:bot,n_0:int,s_1:int) -> int :
  if n_0 == 0:
    return s_1
  else:
    return (let clos_2 = (tail_sum,)
            in clos_2[0](clos_2, n_0 - 1, n_0 + s_1))

def main() -> int :
  print((let clos_4 = (tail_sum,)
         in clos_4[0](clos_4, 3, 0)) + 36)
  return 0
```

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 `Call(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`.

```
def add1(x : int) -> int:
  return x + 1

y = input_int()
g : Callable[[int], int] = lambda x: x - y
f = add1 if input_int() == 0 else g
print( 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 29 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.

```
y = input_int()
f : Callable[[int],int] = lambda x: x + y
print( f(21) )
```

Closure conversion compiles the application `f(21)` into an indirect call:

```
def lambda_3(fvs_4:tuple[bot,tuple[int]], x_2:int) -> int:
  y_1 = fvs_4[1]
  return x_2 + y_1[0]

def main() -> int:
  y_1 = (777,)
  y_1[0] = input_int()
  f_0 = (lambda_3, y_1)
  print((let clos_5 = f_0 in clos_5[0](clos_5, 21)))
  return 0
```

but we can instead compile the application `f(21)` into a direct call:

```
def main() -> int:
  y_1 = (777,)
  y_1[0] = input_int()
  f_0 = (lambda_3, y_1)
  print(lambda_3(f_0, 21))
  return 0
```

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 previous assignment to the variable is 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 assignment, mapping the variable to the name of the global function for the closure. This pass should come after closure conversion.

Exercise 30 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).

8.11 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 lookup 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 lookup 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 discuss the compilation of $\mathcal{L}_{\mathsf{Dyn}}$, a dynamically typed language that is a subset of Python. The dynamic typing is in contrast to the previous chapters, which have studied the compilation of statically typed languages. In dynamically typed languages such as $\mathcal{L}_{\mathsf{Dyn}}$, 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 (False if input_int() == 1 else 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 "morphos", 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 Python, the `not` operator produces a result for any kind of value: given `False` it returns `True` and given anything else it returns `False`.

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 `True[0]` results in a run-time error because the first argument must be a tuple, not a Boolean.

The concrete and abstract syntax of $\mathcal{L}_{\mathsf{Dyn}}$ is defined in Figures 9.1 and 9.2. There is no type checker for $\mathcal{L}_{\mathsf{Dyn}}$ because dynamically typed languages check types at runtime.

The definitional interpreter for $\mathcal{L}_{\mathsf{Dyn}}$ is presented in Figures 9.3 and 9.4 and its auxiliary functions are defined in Figure 9.5. Consider the match case for `Constant(n)`. Instead of simply returning the integer `n` (as in the interpreter for $\mathcal{L}_{\mathsf{Var}}$ in Figure 2.4), the interpreter for $\mathcal{L}_{\mathsf{Dyn}}$ creates a *tagged value* that combines an underlying value with a tag that identifies what kind of value it is. We define the following class to represented tagged values.

$$
\begin{array}{rcl}
\mathit{cmp} & ::= & \texttt{==} \mid \texttt{!=} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{is} \\
\mathit{exp} & ::= & \mathit{int} \mid \texttt{input\_int()} \mid \texttt{-}\mathit{exp} \mid \mathit{exp}\,\texttt{+}\,\mathit{exp} \mid \mathit{exp}\,\texttt{-}\,\mathit{exp} \mid \texttt{(}\mathit{exp}\texttt{)} \\
& \mid & \mathit{var} \mid \texttt{True} \mid \texttt{False} \mid \mathit{exp}\ \texttt{and}\ \mathit{exp} \mid \mathit{exp}\ \texttt{or}\ \mathit{exp} \mid \texttt{not}\ \mathit{exp} \\
& \mid & \mathit{exp}\ \mathit{cmp}\ \mathit{exp} \mid \mathit{exp}\ \texttt{if}\ \mathit{exp}\ \texttt{else}\ \mathit{exp} \\
& \mid & \mathit{exp}\texttt{,} \ldots \texttt{,}\mathit{exp} \mid \mathit{exp}\texttt{[}\mathit{exp}\texttt{]} \mid \texttt{len(}\mathit{exp}\texttt{)} \\
& \mid & \mathit{exp}\texttt{(}\mathit{exp}\texttt{,} \ldots\texttt{)} \mid \texttt{lambda}\,\mathit{var}\texttt{,}\ \ldots\ \texttt{:}\ \mathit{exp} \\
\mathit{stmt} & ::= & \texttt{print(}\mathit{exp}\texttt{)} \mid \mathit{exp} \mid \mathit{var}\,\texttt{=}\,\mathit{exp} \\
& \mid & \texttt{if}\ \mathit{exp}\texttt{:}\ \mathit{stmt}^{+}\ \texttt{else:}\ \mathit{stmt}^{+} \mid \texttt{while}\ \mathit{exp}\texttt{:}\ \mathit{stmt}^{+} \\
& \mid & \texttt{return}\ \mathit{exp} \\
\mathit{def} & ::= & \texttt{def}\ \mathit{var}\texttt{(}\mathit{var}\texttt{,}\ldots\texttt{):}\ \mathit{stmt}^{+} \\
\mathcal{L}_{\mathsf{Dyn}} & ::= & \mathit{def}\ldots\mathit{stmt}\ldots
\end{array}
$$

Figure 9.1
Syntax of $\mathcal{L}_{\mathsf{Dyn}}$, an untyped language (a subset of Python).

$$
\begin{array}{rcl}
\mathit{binaryop} & ::= & \texttt{Add()} \mid \texttt{Sub()} \\
\mathit{unaryop} & ::= & \texttt{USub()} \mid \texttt{Not()} \\
\mathit{boolop} & ::= & \texttt{And()} \mid \texttt{Or()} \\
\mathit{cmp} & ::= & \texttt{Eq()} \mid \texttt{NotEq()} \mid \texttt{Lt()} \mid \texttt{LtE()} \mid \texttt{Gt()} \mid \texttt{GtE()} \\
& \mid & \texttt{Is()} \\
\mathit{bool} & ::= & \texttt{True} \mid \texttt{False} \\
\mathit{exp} & ::= & \texttt{Constant(}\mathit{int}\texttt{)} \mid \texttt{Call(Name('input\_int'),[])} \\
& \mid & \texttt{UnaryOp(}\mathit{unaryop}\texttt{,}\mathit{exp}\texttt{)} \mid \texttt{BinOp(}\mathit{exp}\texttt{,}\mathit{binaryop}\texttt{,}\mathit{exp}\texttt{)} \mid \texttt{Name(}\mathit{var}\texttt{)} \\
& \mid & \texttt{Constant(}\mathit{bool}\texttt{)} \mid \texttt{BoolOp(}\mathit{boolop}\texttt{,[}\mathit{exp}\texttt{,}\mathit{exp}\texttt{])} \\
& \mid & \texttt{Compare(}\mathit{exp}\texttt{,[}\mathit{cmp}\texttt{],[}\mathit{exp}\texttt{])} \mid \texttt{IfExp(}\mathit{exp}\texttt{,}\mathit{exp}\texttt{,}\mathit{exp}\texttt{)} \\
& \mid & \texttt{Tuple(}\mathit{exp}^{+}\texttt{,Load())} \mid \texttt{Subscript(}\mathit{exp}\texttt{,}\mathit{exp}\texttt{,Load())} \\
& \mid & \texttt{Call(Name('len'),[}\mathit{exp}\texttt{])} \\
& \mid & \texttt{Call(}\mathit{exp}\texttt{,}\ \mathit{exp}^{*}\texttt{)} \mid \texttt{Lambda(}\mathit{var}^{*}\texttt{,}\ \mathit{exp}\texttt{)} \\
\mathit{stmt} & ::= & \texttt{Expr(Call(Name('print'),[}\mathit{exp}\texttt{]))} \mid \texttt{Expr(}\mathit{exp}\texttt{)} \\
& \mid & \texttt{Assign([Name(}\mathit{var}\texttt{)],}\ \mathit{exp}\texttt{)} \\
& \mid & \texttt{If(}\mathit{exp}\texttt{,}\ \mathit{stmt}^{+}\texttt{,}\ \mathit{stmt}^{+}\texttt{)} \mid \texttt{While(}\mathit{exp}\texttt{,}\ \mathit{stmt}^{+}\texttt{, [])} \\
& \mid & \texttt{Return(}\mathit{exp}\texttt{)} \\
\mathit{params} & ::= & \texttt{(}\mathit{var}\texttt{,AnyType())}^{*} \\
\mathit{def} & ::= & \texttt{FunctionDef(}\mathit{var}\texttt{,}\ \mathit{params}\texttt{,}\ \mathit{stmt}^{+}\texttt{, None, AnyType(), None)} \\
\mathcal{L}_{\mathsf{Dyn}} & ::= & \texttt{Module([}\mathit{def}\ldots\mathit{stmt}\ldots\texttt{])}
\end{array}
$$

Figure 9.2
The abstract syntax of $\mathcal{L}_{\mathsf{Dyn}}$.

```
@dataclass(eq=True)
class Tagged(Value):
  value : Value
  tag : str
  def __str__(self):
    return str(self.value)
```

The tags are `'int'`, `'bool'`, `'none'`, `'tuple'`, and `'function'`. Tags are closely related to types but don't always capture all the information that a type does. For example, a tuple of type `TupleType([AnyType(),AnyType()])` is tagged with `'tuple'` and a function of type `FunctionType([AnyType(), AnyType()], AnyType())` is tagged with `'function'`.

Next consider the match case for accessing the element of a tuple. The `untag` auxiliary function (Figure 9.5) is used to ensure that the first argument is a tuple and the second is an integer. If they are not, an exception is raised. The compiled code must also signal an error by exiting with return code `255`. A exception is also raised if the index is not less than the length of the tuple or if it is negative.

```
class InterpLdyn(InterpLlambda):

  def interp_exp(self, e, env):
    match e:
      case Constant(n):
        return self.tag(super().interp_exp(e, env))
      case Tuple(es, Load()):
        return self.tag(super().interp_exp(e, env))
      case Lambda(params, body):
        return self.tag(super().interp_exp(e, env))
      case Call(Name('input_int'), []):
        return self.tag(super().interp_exp(e, env))
      case BinOp(left, Add(), right):
          l = self.interp_exp(left, env); r = self.interp_exp(right, env)
          return self.tag(self.untag(l, 'int', e) + self.untag(r, 'int', e))
      case BinOp(left, Sub(), right):
          l = self.interp_exp(left, env); r = self.interp_exp(right, env)
          return self.tag(self.untag(l, 'int', e) - self.untag(r, 'int', e))
      case UnaryOp(USub(), e1):
          v = self.interp_exp(e1, env)
          return self.tag(- self.untag(v, 'int', e))
      case IfExp(test, body, orelse):
        v = self.interp_exp(test, env)
        if self.untag(v, 'bool', e):
            return self.interp_exp(body, env)
        else:
            return self.interp_exp(orelse, env)
      case UnaryOp(Not(), e1):
        v = self.interp_exp(e1, env)
        return self.tag(not self.untag(v, 'bool', e))
      case BoolOp(And(), values):
        left = values[0]; right = values[1]
        l = self.interp_exp(left, env)
        if self.untag(l, 'bool', e):
            return self.interp_exp(right, env)
        else:
            return self.tag(False)
      case BoolOp(Or(), values):
        left = values[0]; right = values[1]
        l = self.interp_exp(left, env)
        if self.untag(l, 'bool', e):
            return self.tag(True)
        else:
            return self.interp_exp(right, env)
      case Compare(left, [cmp], [right]):
        l = self.interp_exp(left, env)
        r = self.interp_exp(right, env)
        if l.tag == r.tag:
          return self.tag(self.interp_cmp(cmp)(l.value, r.value))
        else:
          raise Exception('interp Compare unexpected ' \
                          + repr(l) + ' ' + repr(r))
      case Subscript(tup, index, Load()):
        t = self.interp_exp(tup, env)
        n = self.interp_exp(index, env)
        return self.untag(t, 'tuple', e)[self.untag(n, 'int', e)]
      case Call(Name('len'), [tup]):
        t = self.interp_exp(tup, env)
        return self.tag(len(self.untag(t, 'tuple', e)))
      case _:
        return self.tag(super().interp_exp(e, env))
```

Figure 9.3
Interpreter for the $\mathcal{L}_{\mathsf{Dyn}}$ language, part 1.

```
class InterpLdyn(InterpLlambda):

  def interp_stmts(self, ss, env):
    if len(ss) == 0:
      return
    match ss[0]:
      case If(test, body, orelse):
        v = self.interp_exp(test, env)
        if self.untag(v, 'bool', ss[0]):
            return self.interp_stmts(body + ss[1:], env)
        else:
            return self.interp_stmts(orelse + ss[1:], env)
      case While(test, body, []):
        while self.untag(self.interp_exp(test, env), 'bool', ss[0]):
            self.interp_stmts(body, env)
        return self.interp_stmts(ss[1:], env)
      case Assign([Subscript(tup, index)], value):
        tup = self.interp_exp(tup, env)
        index = self.interp_exp(index, env)
        tup_v = self.untag(tup, 'tuple', ss[0])
        index_v = self.untag(index, 'int', ss[0])
        tup_v[index_v] = self.interp_exp(value, env)
        return self.interp_stmts(ss[1:], env)
      case FunctionDef(name, params, bod, dl, returns, comment):
        ps = [x for (x,t) in params]
        env[name] = self.tag(Function(name, ps, bod, env))
        return self.interp_stmts(ss[1:], env)
      case _:
        return super().interp_stmts(ss, env)
```

Figure 9.4
Interpreter for the $\mathcal{L}_{\mathsf{Dyn}}$ language, part 2.

```
class InterpLdyn(InterpLlambda):

  def tag(self, v):
      if v is True or v is False:
          return Tagged(v, 'bool')
      elif isinstance(v, int):
          return Tagged(v, 'int')
      elif isinstance(v, Function):
          return Tagged(v, 'function')
      elif isinstance(v, tuple):
          return Tagged(v, 'tuple')
      elif isinstance(v, type(None)):
          return Tagged(v, 'none')
      else:
          raise Exception('tag: unexpected ' + repr(v))

  def untag(self, v, expected_tag, ast):
      match v:
        case Tagged(val, tag) if tag == expected_tag:
          return val
        case _:
          raise Exception('expected Tagged value with ' + expected_tag + ', not ' + ' ' + repr(v))

  def apply_fun(self, fun, args, e):
      f = self.untag(fun, 'function', e)
      return super().apply_fun(f, args, e)
```

Figure 9.5
Auxiliary functions for the $\mathcal{L}_{\mathsf{Dyn}}$ interpreter.

9.1 Representation of Tagged Values

The interpreter for $\mathcal{L}_{\mathsf{Dyn}}$ introduced a new kind of value, a tagged value. To compile $\mathcal{L}_{\mathsf{Dyn}}$ to x86 we must decide how to represent tagged values at the bit level. Because almost every operation in $\mathcal{L}_{\mathsf{Dyn}}$ 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 vectors, 011 for procedures, and 101 for the void value, `None`. We define the following auxiliary function for mapping types to tag codes.

$$
\begin{aligned}
\mathit{tagof}(\texttt{IntType()}) &= 001\\
\mathit{tagof}(\texttt{BoolType()}) &= 100\\
\mathit{tagof}(\texttt{TupleType(ts)}) &= 010\\
\mathit{tagof}(\texttt{FunctionType(ps, rt)}) &= 011\\
\mathit{tagof}(\texttt{type(None)}) &= 101
\end{aligned}
$$

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 vectors and procedures because those values are addresses, and our addresses are 8-byte aligned so the rightmost 3 bits are unused, they are always 000. Thus, we do not lose information by overwriting the rightmost 3 bits with the tag and we can simply zero-out the tag to recover the original address.

To make tagged values into first-class entities, we can give them a type, called `AnyType()`, and define operations such as `Inject` and `Project` for creating and using them, yielding the $\mathcal{L}_{\mathsf{Any}}$ intermediate language. We describe how to compile $\mathcal{L}_{\mathsf{Dyn}}$ to $\mathcal{L}_{\mathsf{Any}}$ in Section 9.3 but first we describe the $\mathcal{L}_{\mathsf{Any}}$ language in greater detail.

9.2 The $\mathcal{L}_{\mathsf{Any}}$ Language

The abstract syntax of $\mathcal{L}_{\mathsf{Any}}$ is defined in Figure 9.6. 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 *ftype*, which simplifies the implementation and corresponds with the needs for compiling $\mathcal{L}_{\mathsf{Dyn}}$.

The operators `any_tuple_load` and `any_len` adapt the tuple operations so that they can be applied to a value of type `AnyType`. 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 checker for $\mathcal{L}_{\mathsf{Any}}$ is shown in Figure 9.7 . The interpreter for $\mathcal{L}_{\mathsf{Any}}$ is in Figure 9.8 and its auxiliary functions are in Figure 9.9.

binaryop	::=	`Add()` | `Sub()`
unaryop	::=	`USub()`
exp	::=	`Constant(`*int*`)` | `Call(Name('input_int'),[])`
	|	`UnaryOp(`*unaryop*`,`*exp*`)` | `BinOp(`*binaryop*`,`*exp*`,`*exp*`)`
stmt	::=	`Expr(Call(Name('print'),[`*exp*`]))` | `Expr(`*exp*`)`
exp	::=	`Name(`*var*`)`
stmt	::=	`Assign([Name(`*var*`)], `*exp*`)`
boolop	::=	`And()` | `Or()`
unaryop	::=	`Not()`
cmp	::=	`Eq()` | `NotEq()` | `Lt()` | `LtE()` | `Gt()` | `GtE()`
bool	::=	`True` | `False`
exp	::=	`Constant(`*bool*`)` | `BoolOp(`*boolop*`,[`*exp*`,`*exp*`])`
	|	`Compare(`*exp*`,[`*cmp*`],[`*exp*`])` | `IfExp(`*exp*`,`*exp*`,`*exp*`)`
stmt	::=	`If(`*exp*`, `$stmt^+$`, `$stmt^+$`)`
stmt	::=	`While(`*exp*`, `$stmt^+$`, [])`
cmp	::=	`Is()`
exp	::=	`Tuple(`exp^+`,Load())` | `Subscript(`*exp*`,Constant(`*int*`),Load())`
	|	`Call(Name('len'),[`*exp*`])`
type	::=	`IntType()` | `BoolType()VoidType()` | `TupleType[`$type^+$`]`
	|	`FunctionType(`$type^*$`, `*type*`)`
exp	::=	`Call(`*exp*`, `exp^*`)`
stmt	::=	`Return(`*exp*`)`
params	::=	$(var,type)^*$
def	::=	`FunctionDef(`*var*`, `*params*`, `$stmt^+$`, None, `*type*`, None)`
exp	::=	`Lambda(`var^*`, `*exp*`)` | `Call(Name('arity'), [`*exp*`])`
stmt	::=	`AnnAssign(`*var*`, `*type*`, `*exp*`, 0)`
type	::=	`AnyType()`
ftype	::=	`IntType()` | `BoolType()` | `VoidType()` | `TupleType[AnyType()`$^+$`]`
	|	`FunctionType(AnyType()`*`, AnyType())`
exp	::=	`Inject(`*exp*`, `*ftype*`)` | `Project(`*exp*`, `*ftype*`)`
	|	`Call(Name('any_tuple_load'), [`*exp*`, Constant(`*n*`)])`
	|	`Call(Name('any_len'), [`*exp*`])`
$\mathcal{L}_{\mathsf{Any}}$	::=	`Module([`*def* … *stmt* …`])`

Figure 9.6
The abstract syntax of $\mathcal{L}_{\mathsf{Any}}$, extending $\mathcal{L}_\lambda$ (Figure 8.4).

```
class TypeCheckLany(TypeCheckLlambda):

  def type_check_exp(self, e, env):
    match e:
      case Inject(value, typ):
        self.check_exp(value, typ, env)
        return AnyType()
      case Project(value, typ):
        self.check_exp(value, AnyType(), env)
        return typ
      case Call(Name('any_tuple_load'), [tup, index]):
        self.check_exp(tup, AnyType(), env)
        return AnyType()
      case Call(Name('any_len'), [tup]):
        self.check_exp(tup, AnyType(), env)
        return IntType()
      case Call(Name('arity'), [fun]):
        ty = self.type_check_exp(fun, env)
        match ty:
          case FunctionType(ps, rt):
            return IntType()
          case TupleType([FunctionType(ps,rs)]):
            return IntType()
          case _:
            raise Exception('type_check_exp arity unexpected ' + repr(ty))
      case Call(Name('make_any'), [value, tag]):
        self.type_check_exp(value, env)
        self.check_exp(tag, IntType(), env)
        return AnyType()
      case ValueOf(value, typ):
        self.check_exp(value, AnyType(), env)
        return typ
      case TagOf(value):
        self.check_exp(value, AnyType(), env)
        return IntType()
      case Call(Name('exit'), []):
        return Bottom()
      case AnnLambda(params, returns, body):
        new_env = {x:t for (x,t) in env.items()}
        for (x,t) in params:
            new_env[x] = t
        return_t = self.type_check_exp(body, new_env)
        self.check_type_equal(returns, return_t, e)
        return FunctionType([t for (x,t) in params], return_t)
      case _:
        return super().type_check_exp(e, env)
```

Figure 9.7
Type checker for the $\mathcal{L}_{\mathsf{Any}}$ language.

```
class InterpLany(InterpLlambda):

  def interp_exp(self, e, env):
    match e:
      case Inject(value, typ):
        v = self.interp_exp(value, env)
        return Tagged(v, self.type_to_tag(typ))
      case Project(value, typ):
        v = self.interp_exp(value, env)
        match v:
          case Tagged(val, tag) if self.type_to_tag(typ) == tag:
            return val
          case _:
            raise Exception('interp project to ' + repr(typ) \
                            + '  unexpected ' + repr(v))
      case Call(Name('any_tuple_load'), [tup, index]):
        tv = self.interp_exp(tup, env)
        n = self.interp_exp(index, env)
        match tv:
          case Tagged(v, tag):
            return v[n]
          case _:
            raise Exception('interp any_tuple_load unexpected ' + repr(tv))
      case Call(Name('any_tuple_store'), [tup, index, value]):
        tv = self.interp_exp(tup, env)
        n = self.interp_exp(index, env)
        val = self.interp_exp(value, env)
        match tv:
          case Tagged(v, tag):
            v[n] = val
            return None
          case _:
            raise Exception('interp any_tuple_load unexpected ' + repr(tv))
      case Call(Name('any_len'), [value]):
        v = self.interp_exp(value, env)
        match v:
          case Tagged(value, tag):
            return len(value)
          case _:
            raise Exception('interp any_len unexpected ' + repr(v))
      case Call(Name('make_any'), [value, tag]):
        v = self.interp_exp(value, env)
        t = self.interp_exp(tag, env)
        return Tagged(v, t)
      case Call(Name('arity'), [fun]):
        f = self.interp_exp(fun, env)
        return self.arity(f)
      case Call(Name('exit'), []):
        trace('exiting!')
        exit(0)
      case TagOf(value):
        v = self.interp_exp(value, env)
        match v:
          case Tagged(val, tag):
            return tag
          case _:
            raise Exception('interp TagOf unexpected ' + repr(v))
      case ValueOf(value, typ):
        v = self.interp_exp(value, env)
        match v:
          case Tagged(val, tag):
            return val
          case _:
            raise Exception('interp ValueOf unexpected ' + repr(v))
      case AnnLambda(params, returns, body):
        return Function('lambda', [x for (x,t) in params], [Return(body)], env)
      case _:
        return super().interp_exp(e, env)
```

Figure 9.8
Interpreter for $\mathcal{L}_{\mathsf{Any}}$.

```
class InterpLany(InterpLlambda):

  def type_to_tag(self, typ):
      match typ:
        case FunctionType(params, rt):
          return 'function'
        case TupleType(fields):
          return 'tuple'
        case t if t == int:
          return 'int'
        case t if t == bool:
          return 'bool'
        case IntType():
          return 'int'
        case BoolType():
          return 'int'
        case _:
          raise Exception('type_to_tag unexpected ' + repr(typ))

  def arity(self, v):
    match v:
      case Function(name, params, body, env):
        return len(params)
      case ClosureTuple(args, arity):
        return arity
      case _:
        raise Exception('Lany arity unexpected ' + repr(v))
```

Figure 9.9
Auxiliary functions for interpreting $\mathcal{L}_{\mathsf{Any}}$.

`True`	$\Rightarrow$	`Inject(True, BoolType())`

e_1 `+` e_2	$\Rightarrow$	`Inject(Project(`e'_1`, IntType())`
		`    + Project(`e'_2`, IntType()),`
		`    IntType())`

`lambda` $x_1 \ldots x_n$`:` e	$\Rightarrow$	`Inject(Lambda([(`x_1`,AnyType),`$\ldots$`,(`x_n`,AnyType)],` e'`)` `    FunctionType([AnyType(),`$\ldots$`], AnyType()))`
e_0`(`$e_1 \ldots e_n$`)`	$\Rightarrow$	`Call(Project(`e'_0`, FunctionType([AnyType(),`$\ldots$`],` `    AnyType())),` $e'_1, \ldots, e'_n$`)`
e_1`[`e_2`]`	$\Rightarrow$	`Call(Name('any_tuple_load'),[`e'_1`,` e'_2`])`

Figure 9.10
Cast Insertion

9.3 Cast Insertion: Compiling $\mathcal{L}_{\mathsf{Dyn}}$ to $\mathcal{L}_{\mathsf{Any}}$

The `cast_insert` pass compiles from $\mathcal{L}_{\mathsf{Dyn}}$ to $\mathcal{L}_{\mathsf{Any}}$. Figure 9.10 shows the compilation of many of the $\mathcal{L}_{\mathsf{Dyn}}$ forms into $\mathcal{L}_{\mathsf{Any}}$. An important invariant of this pass is that given a subexpression e in the $\mathcal{L}_{\mathsf{Dyn}}$ program, the pass will produce an expression e' in $\mathcal{L}_{\mathsf{Any}}$ that has type `AnyType`. For example, the first row in Figure 9.10 shows the compilation of the Boolean `True`, which must be injected to produce an expression of type `AnyType`. The second row of Figure 9.10, the compilation of addition, is representative of compilation for many primitive operations: the arguments have type `AnyType` and must be projected to `IntType` 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 `AnyType`.

9.4 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 the `exit` function (from the C standard library) and two new AST classes: `TagOf` and `ValueOf`. The `exit` function ends the execution of the program. The `TagOf` operation retrieves the type tag from a tagged value of type `AnyType`. The `ValueOf` operation retrieves the underlying value from a tagged value. The `ValueOf` operation includes the type for the underlying value which is used by the type checker.

If the target type of the projection is `bool` or `int`, then `Project` can be translated as follows.

```
Project(e, ftype)
⇒
Begin([Assign([tmp], e′)],
      IfExp(Compare(TagOf(tmp),[Eq()],
                    [Constant(tagof(ftype))]),
            ValueOf(tmp, ftype)
            Call(Name('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)
⇒
Call(Name('make_any'), [e′, Constant(tagof(ftype))])
```

The introduction of `make_any` makes it difficult to use bidirectional type checking because we no longer have an expected type to use for type checking the expression e'. Thus, we run into difficulty if e' is a `Lambda` expression. We recommend translating `Lambda` to a new AST class `AnnLambda` (for annotated lambda) whose parameters have type annotations and that records the return type.

The `any_tuple_load` operation combines the projection action with the load operation. Also, the load operation allows arbitrary expressions for the index so the type checker for $\mathcal{L}_{\mathsf{Any}}$ (Figure 9.7) cannot guarantee that the index is within bounds. Thus, we insert code to perform bounds checking at runtime. The translation for `any_tuple_load` is as follows.

```
Call(Name('any_tuple_load'), [e1,e2])
⇒
Block([Assign([t], e′1), Assign([i], e′2)],
     IfExp(Compare(TagOf(t), [Eq()], [Constant(2)]),
          IfExp(Compare(i, [Lt()], [Call(Name('any_len'), [t])]),
               Call(Name('any_tuple_load'), [t, i]),
               Call(Name('exit'), [])),
          Call(Name('exit'), [])))
```

9.5 Assignment Conversion

Update this pass to handle the `TagOf`, `ValueOf`, and `AnnLambda` AST classes.

9.6 Closure Conversion

Update this pass to handle the `TagOf`, `ValueOf`, and `AnnLambda` AST classes.

$$
\begin{array}{lcl}
\mathit{atm} & ::= & \texttt{Constant(}\mathit{int}\texttt{)} \mid \texttt{Name(}\mathit{var}\texttt{)} \mid \texttt{Constant(}\mathit{bool}\texttt{)} \\
\mathit{exp} & ::= & \mathit{atm} \mid \texttt{Call(Name('input\_int'),[])} \\
& \mid & \texttt{BinOp(}\mathit{atm}\texttt{,}\mathit{binaryop}\texttt{,}\mathit{atm}\texttt{)} \mid \texttt{UnaryOp(}\mathit{unaryop}\texttt{,}\mathit{atm}\texttt{)} \\
& \mid & \texttt{Compare(}\mathit{atm}\texttt{,[}\mathit{cmp}\texttt{],[}\mathit{atm}\texttt{])} \\
\mathit{stmt} & ::= & \texttt{Expr(Call(Name('print'),[}\mathit{atm}\texttt{]))} \mid \texttt{Expr(}\mathit{exp}\texttt{)} \\
& \mid & \texttt{Assign([Name(}\mathit{var}\texttt{)], }\mathit{exp}\texttt{)} \mid \texttt{Return(}\mathit{exp}\texttt{)} \mid \texttt{Goto(}\mathit{label}\texttt{)} \\
& \mid & \texttt{If(Compare(}\mathit{atm}\texttt{,[}\mathit{cmp}\texttt{],[}\mathit{atm}\texttt{]), [Goto(}\mathit{label}\texttt{)], [Goto(}\mathit{label}\texttt{)])} \\
\hline
\mathit{exp} & ::= & \texttt{Subscript(}\mathit{atm}\texttt{,}\mathit{atm}\texttt{,Load())} \mid \texttt{Allocate(}\mathit{int}, \mathit{type}\texttt{)} \\
& \mid & \texttt{GlobalValue(}\mathit{var}\texttt{)} \mid \texttt{Call(Name('len'),[}\mathit{atm}\texttt{])} \\
\mathit{stmt} & ::= & \texttt{Collect(}\mathit{int}\texttt{)} \\
& \mid & \texttt{Assign([Subscript(}\mathit{atm}\texttt{,}\mathit{atm}\texttt{,Store())], }\mathit{atm}\texttt{)} \\
\hline
\mathit{exp} & ::= & \texttt{FunRef(}\mathit{label}\texttt{, }\mathit{int}\texttt{)} \mid \texttt{Call(}\mathit{atm}\texttt{, }\mathit{atm}^{*}\texttt{)} \\
\mathit{stmt} & ::= & \texttt{TailCall(}\mathit{atm}\texttt{, }\mathit{atm}^{*}\texttt{)} \\
\mathit{params} & ::= & \texttt{[(}\mathit{var}\texttt{,}\mathit{type}\texttt{), } \ldots \texttt{]} \\
\mathit{block} & ::= & \mathit{label}\texttt{:}\mathit{stmt}^{*} \\
\mathit{blocks} & ::= & \texttt{\{}\mathit{block}\texttt{, } \ldots \texttt{\}} \\
\mathit{def} & ::= & \texttt{FunctionDef(}\mathit{label}\texttt{, }\mathit{params}\texttt{, }\mathit{blocks}\texttt{, None, }\mathit{type}\texttt{, None)} \\
\hline
\mathit{exp} & ::= & \texttt{Uninitialized(}\mathit{type}\texttt{)} \mid \texttt{AllocateClosure(}\mathit{len}, \mathit{type}, \mathit{arity}\texttt{)} \\
& \mid & \texttt{Call(Name('arity'), [}\mathit{atm}\texttt{])} \\
\hline
\mathit{exp} & ::= & \texttt{Call(Name('make\_any'), [}\mathit{atm}, \mathit{atm}\texttt{])} \\
& \mid & \texttt{TagOf(}\mathit{atm}\texttt{)} \mid \texttt{ValueOf(}\mathit{atm}, \mathit{ftype}\texttt{)} \\
& \mid & \texttt{Call(Name('any\_tuple\_load'), [}\mathit{atm}, \mathit{atm}\texttt{])} \\
& \mid & \texttt{Call(Name('any\_tuple\_store'), [}\mathit{atm}, \mathit{atm}, \mathit{atm}\texttt{])} \\
& \mid & \texttt{Call(Name('any\_len'), [}\mathit{atm}\texttt{])} \\
& \mid & \texttt{Call(Name('exit'), [])} \\
\mathcal{C}_{\mathsf{Any}} & ::= & \texttt{CProgramDefs([}\mathit{def}\texttt{, } \ldots \texttt{])}
\end{array}
$$

Figure 9.11
The abstract syntax of $\mathcal{C}_{\mathsf{Any}}$, extending $\mathcal{C}_{\mathsf{Clos}}$ (Figure 8.9).

9.7 Remove Complex Operands

The `ValueOf` and `TagOf` operations are both complex expressions. Their subexpressions must be atomic.

9.8 Explicate Control and $\mathcal{C}_{\mathsf{Any}}$

The output of `explicate_control` is the $\mathcal{C}_{\mathsf{Any}}$ language whose syntax is defined in Figure 9.11. Update the auxiliary functions `explicate_tail`, `explicate_effect`, and `explicate_pred` as appropriately to handle the new expressions in $\mathcal{C}_{\mathsf{Any}}$.

9.9 Select Instructions

In the `select_instructions` pass we translate the primitive operations on the `AnyType` 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 `int` or `bool`. 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], Call(Name('make_any'), [e, Constant(tag)]))
⇒
movq e', lhs'
salq $3, lhs'
orq $tag, lhs'
```

The instruction selection for tuples and procedures is different because their 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`.

```
Assign([lhs], Call(Name('make_any'), [e, Constant(tag)]))
⇒
movq e', lhs'
orq $tag, lhs'
```

`TagOf` Recall that the `TagOf` operation extracts the type tag from a value of type `AnyType`. 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], TagOf(e))
⇒
movq e', lhs'
andq $7, lhs'
```

`ValueOf` Like `make_any`, the instructions for `ValueOf` are different 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 just need to zero-out the rightmost 3 bits. We accomplish this by creating the bit pattern $\ldots 0111$ (7 in decimal) and apply bitwise-not to obtain $\ldots 11111000$ (-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_len` The `any_len` 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], Call(Name('any_len'), [e1]))
⟹
movq $-8, %r11
andq e'1, %r11
movq 0(%r11), %r11
andq $126, %r11
sarq $1, %r11
movq %r11, lhs'
```

`any_tuple_load` This operation combines the effect of `ValueOf` with reading an element of the tuple (see Section 6.6). 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], Call(Name('any_tuple_load'), [e1,e2]))
⟹
movq $-8, %r11
andq e'1, %r11
movq e'2, %rax
addq $1, %rax
imulq $8, %rax
addq %rax, %r11
movq 0(%r11) lhs'
```

`any_tuple_store` The code generation for `any_tuple_store` is analogous to the above translation for reading from a tuple.

9.10 Register Allocation for $\mathcal{L}_{\mathsf{Any}}$

There is an interesting interaction between tagged values and garbage collection that has an impact on register allocation. A variable of type `AnyType` 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 `AnyType` in a similar way to variables of tuple type for purposes of register allocation. In particular,

- If a variable of type `AnyType` 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 `AnyType` that are live after a `callq` as interfering with all the registers.
- If a variable of type `AnyType` 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 31 Expand your compiler to handle $\mathcal{L}_{\mathsf{Dyn}}$ as outlined in this chapter. Create tests for $\mathcal{L}_{\mathsf{Dyn}}$ 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 $\mathcal{L}_{\mathsf{Dyn}}$ programs that run to completion without error.

Figure 9.12 provides an overview of all the passes needed for the compilation of $\mathcal{L}_{\mathsf{Dyn}}$.

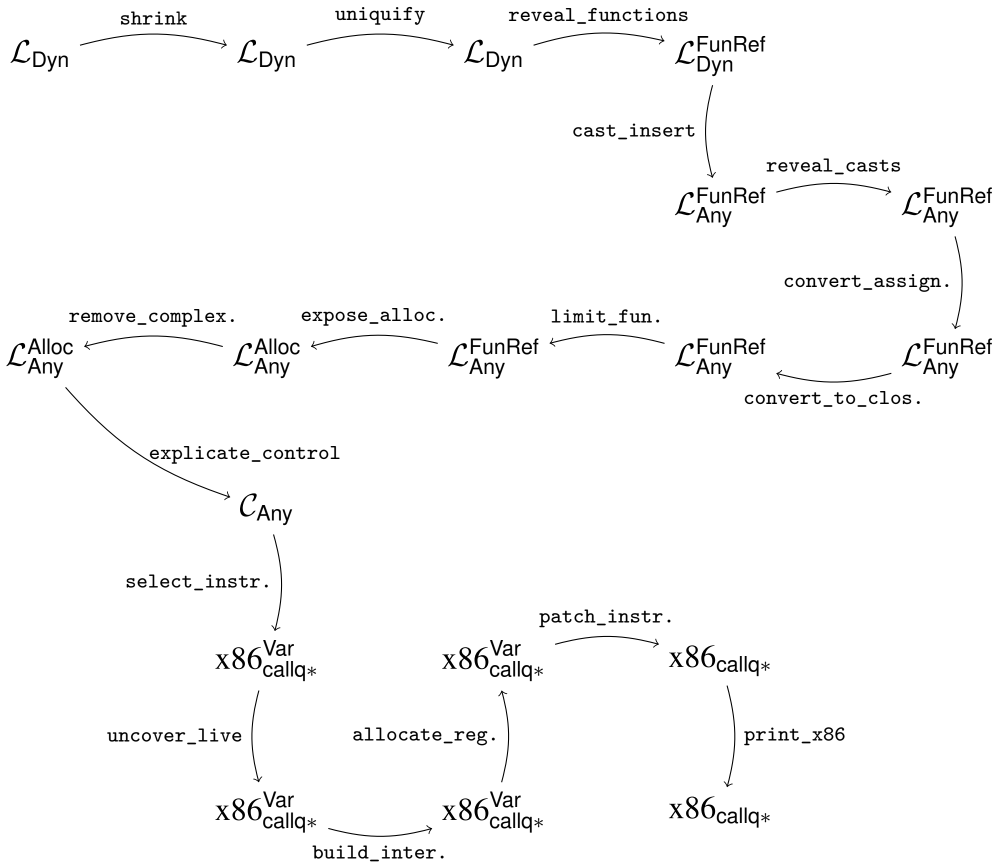


Figure 9.12

Diagram of the passes for $\mathcal{L}_{\mathsf{Dyn}}$, a dynamically typed language.

10 Gradual Typing

UNDER CONSTRUCTION

11 Parametric Polymorphism

UNDER CONSTRUCTION

A Appendix

A.1 x86 Instruction Set Quick-Reference

Table A.1 lists some x86 instructions and what they do. We write $A \to B$ to mean that the value of A is written into location B. Address offsets are given in bytes. The instruction arguments A, B, C can be immediate constants (such as `$4`), registers (such as `%rax`), or memory references (such as `-4(%ebp)`). Most x86 instructions only allow at most one memory reference per instruction. Other operands must be immediates or registers.

Instruction	Operation
`addq` A, B	$A + B \to B$
`negq` A	$-A \to A$
`subq` A, B	$B - A \to B$
`imulq` A, B	$A \times B \to B$
`callq` L	Pushes the return address and jumps to label L
`callq` *A	Calls the function at the address A.
`retq`	Pops the return address and jumps to it
`popq` A	$*\texttt{rsp} \to A$; $\texttt{rsp} + 8 \to \texttt{rsp}$
`pushq` A	$\texttt{rsp} - 8 \to \texttt{rsp}$; $A \to *\texttt{rsp}$
`leaq` A,B	$A \to B$ (B must be a register)
`cmpq` A, B	compare A and B and set the flag register (B must not be an immediate)
`je` L `jl` L `jle` L `jg` L `jge` L	Jump to label L if the flag register matches the condition code of the instruction, otherwise go to the next instructions. The condition codes are `e` for "equal", `l` for "less", `le` for "less or equal", `g` for "greater", and `ge` for "greater or equal".
`jmp` L	Jump to label L
`movq` A, B	$A \to B$
`movzbq` A, B	$A \to B$, where A is a single-byte register (e.g., `al` or `cl`), B is a 8-byte register, and the extra bytes of B are set to zero.
`notq` A	$\sim A \to A$ (bitwise complement)
`orq` A, B	$A \mid B \to B$ (bitwise-or)
`andq` A, B	$A \& B \to B$ (bitwise-and)
`salq` A, B	$B \ll A \to B$ (arithmetic shift left, where A is a constant)
`sarq` A, B	$B \gg A \to B$ (arithmetic shift right, where A is a constant)
`sete` A `setl` A `setle` A `setg` A `setge` A	If the flag matches the condition code, then $1 \to A$, else $0 \to A$. Refer to `je` above for the description of the condition codes. A must be a single byte register (e.g., `al` or `cl`).

Table 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. doi: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. doi: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. doi: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. doi: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. doi: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. doi: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. doi: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. doi: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 Algoirthm." *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. doi: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.

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. doi: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. doi: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. doi:10.1007/3-540-48743-3_5. https://doi.org/10.1007/3-540-48743-3%5C_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. doi: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. doi: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. doi: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. doi: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). doi: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. 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. doi: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. doi: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. doi: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. doi: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. doi: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. doi: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. doi:10.1145/321312.321317. 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. doi:10.1145/3315573.3329983. 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.

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. "Revised^5 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. doi:10.1145/355588.365140. http://doi.acm.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. doi:10.1145/158511.158613. 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. doi: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. doi: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 ALGORITHMS††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. doi:https://doi.org/10.1016/B978-1-4832-3187-7.50015-5. http://www.sciencedirect.com/science/article/pii/%20B9781483231877500155.

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. doi: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. doi: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. doi: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 AndréL.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 Hriţcu, 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. doi: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.

CPython github repository. 2021. https://github.com/python/cpython. Python Software Foundation.

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. doi: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. doi: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. doi: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 DYBVIG, MATTHEW FLATT, ANTON VAN STRAATEN, ROBBY FINDLER, and JACOB MATTHEWS. 2009. "Revised^6 Report on the Algorithmic Language Scheme." *Journal of Functional Programming* 19 (August): 1–301. ISSN: 1469-7653. doi: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. doi: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. doi: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. doi: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