A Problem Course in Compilation: From Python to x86 Assembly

Revised August 26, 2012

Jeremy G. Siek

Bor-Yuh Evan Chang

UNIVERSITY OF COLORADO BOULDER
E-mail address: {jeremy.siek, evan.chang}@colorado.edu

ABSTRACT. The primary goal of this course is to help students acquire an understanding of what happens "behind the scenes" when programs written in high-level languages are executed on modern hardware. This understanding is useful for estimating performance tradeoffs and debugging programs. A secondary goal of the course is to teach students the principles, techniques, and tools that are used in compiler construction. While an engineer seldom needs to implement a compiler for a general purpose language, it is quite common to implement small domain-specific languages as part of a larger software system. A tertiary goal of this course is to give the students a chance to practice building a large piece of software using good software engineering practices, in particular, using incremental and test-first development.

The general outline of the course is to implement a sequence of compilers for increasingly larger subsets of the Python 2.5 programming language. The subsets are chosen primarily to bring out interesting principles while keeping busy work to a minimum. Nevertheless, the assignments are challenging and require a significant amount of code. The target language for this sequence of compilers is the x86 assembly language, the native language of most personal computers. These notes are organized as a *problem course*, which means that they present the important ideas and pose exercises that incrementally build a compiler, but many details are left to the student to discover.

Contents

CHAPTER 1

Integers and variables

The main concepts in this chapter are

abstract syntax trees: Inside the compiler we represent programs with a data-structure called an abstract syntax tree.

recursive functions: We analyze and manipulate abstract syntax trees with recursive functions.

flattening expressions into instructions: One step in compiling high-level languages to low-level languages is flattening expressions *trees* into *lists* of instructions.

selecting instructions: The x86 assembly language offers a peculiar variety of instructions, so selecting which instructions are needed to get the job done is not always easy.

test-driven development: A compiler is a large piece of software. To maximize our productivity (and minimize bugs!) we use good software engineering practices, such as writing lots of good test cases before writing code.

1.1. ASTs and the P_0 subset of Python

The first subset of Python that we consider is extremely simple: it consists of print statements, assignment statements, some integer arithmetic, and the `input()` function. We call this subset P_0. The following is an example P_0 program.

```
print - input() + input()
```

Programs are written one character at a time but we, as programmers, do not think of programs as sequences of characters. We think about programs in chunks like `if` statements and `for` loops. These chunks often have parts, for example, an `if` statement has a then-clause and an else-clause. Inside the compiler, we often traverse over a program one chunk at a time, going from parent chunks to their children. A data-structure that facilitates this kind of traversal is a *tree*. Each node in the tree represents a programming language construct and each node has edges that point to its children. When a tree is used to represent a program, we call it an *abstract syntax tree*

(AST). While trees in nature grow up with their leaves at the top, we think of ASTs as growing down with the leaves at the bottom.

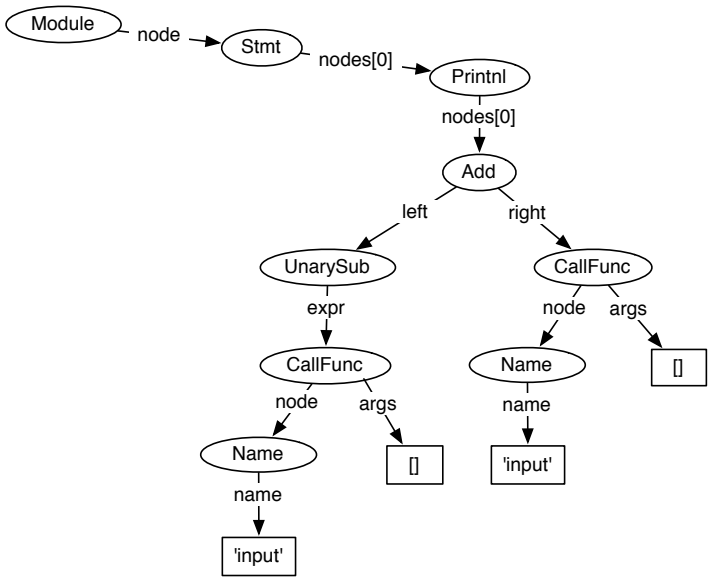


FIGURE 1. The abstract syntax tree for `print - input() + input()`.

Figure 1 shows the abstract syntax tree for the previous P_0 program. There is a standard Python library that turns a sequence of characters into an AST, using a process called *parsing* which we learn about in the next chapter. The following interaction with the Python interpreter shows a call to the Python parser.

```
>>> import compiler
>>> ast = compiler.parse("print - input() + input()")
>>> ast
Module(None,
       Stmt([Printnl([Add((UnarySub(CallFunc(Name('input'),
                                             [], None, None)),
                          CallFunc(Name('input'),
                                   [], None, None)))],
                     None)]))
```

Each node in the AST is a Python object. The objects are instances of Python classes; there is one class for each langauge construct. In the above interaction we invoked `compiler.parse`, but in your compiler I recommend using an alternative function that takes it's input from a file: `compiler.parseFile`. Figure 2 shows the Python classes for the AST nodes for P_0. For each class, we have only listed its constructor, the `__init__` method. You can find out more about these classes by

reading the file `compiler/ast.py` of the Python installation on your computer.

To keep things simple, we place some restrictions on P_0. In a `print` statement, instead of multiple things to print, as in Python 2.5, you only need to support printing one thing. So you can assume the `nodes` attribute of a `Printnl` node is a list containing a single AST node. Similarly, for expression statements you only need to support a single expression, so you do not need to support tuples. P_0 only includes basic assignments instead of the much more general forms supported by Python 2.5. You only need to support a single variable on the left-hand-side. So the `nodes` attribute of `Assign` is a list containing a single `AssName` node whose `flag` attribute is `OP_ASSIGN`. The only kind of value allowed inside of a `Const` node is an integer. P_0 does not include support for Boolean values, so a P_0 AST will never have a `Name` node whose name attribute is "True" or "False".

```
class Module(Node):
    def __init__(self, doc, node):
        self.doc = doc
        self.node = node
class Stmt(Node):
    def __init__(self, nodes):
        self.nodes = nodes
class Printnl(Node):
    def __init__(self, nodes, dest):
        self.nodes = nodes
        self.dest = dest
class Assign(Node):
    def __init__(self, nodes, expr):
        self.nodes = nodes
        self.expr = expr
class AssName(Node):
    def __init__(self, name, flags):
        self.name = name
        self.flags = flags
class Discard(Node):
    def __init__(self, expr):
        self.expr = expr
```

```
class Const(Node):
    def __init__(self, value):
        self.value = value
class Name(Node):
    def __init__(self, name):
        self.name = name
class Add(Node):
    def __init__(self, (left, right)):
        self.left = left
        self.right = right
class UnarySub(Node):
    def __init__(self, expr):
        self.expr = expr
# CallFunc is for calls to the 'input' function
class CallFunc(Node):
    def __init__(self, node, args):
        self.node = node
        self.args = args
```

FIGURE 2. The Python classes for representing P_0 ASTs.

1.2. Understand the meaning of P_0

The *meaning* of Python programs, that is, what happens when you run a program, is defined in the Python Reference Manual **[20]**.

Exercise 1.1. Read the sections of the Python Reference Manual that apply to P_0: 3.2, 5.5, 5.6, 6.1, 6.2, and 6.6. Also read the entry for the `input` function in the Python Library Reference, in section 2.1.

Sometimes it is difficult to understand the technical jargon in programming language reference manuals. A complementary way to learn about the meaning of Python programs is to experiment with the standard Python interpreter. If there is an aspect of the language that you do not understand, create a program that uses that aspect and run it! Suppose you are not sure about a particular feature but have a guess, a *hypothesis*, about how it works. Think of a program that will produce one output if your hypothesis is correct and produce a different output if your hypothesis is incorrect. You can then run the Python interpreter to validate or disprove your hypothesis.

For example, suppose that you are not sure what happens when the result of an arithmetic operation results in a very large integer, an integer too large to be stored in a machine register ($> 2^{31} - 1$). In the language C, integer operations wrap around, so 2×2^{30} produces -2147483648 **[13]**. Does the same thing happen in Python? Let us try it and see:

```
>>> 2 * 2**30
2147483648L
```

No, the number does not wrap around! Instead, Python has two kinds of integers: *plain integers* for integers in the range -2^{31} to $2^{31}-1$ and *long integers* for integers in a range that is only limited by the amount of (virtual) memory in your computer. For P_0 we restrict our attention to just plain integers and say that operations that result in integers outside of the range -2^{31} to $2^{31} - 1$ are undefined.

The built-in Python function `input()` reads in a line from standard input (stdin) and then interprets the string as if it were a Python expression, using the built-in `eval` function. For P_0 we only require a subset of this functionality. The `input` function need only deal with integer literals. A call to the input function, of the form `"input()"`, is parsed into the function call AST node `CallFunc`. You do not need to handle general function calls, just recognize the special case of a function call where the function being called is named `"input"`.

Exercise 1.2. Write some programs in the P_0 subset of Python. The programs should be chosen to help you understand the language. Look for corner cases or unusual aspects of the language to test in your programs. Later in this assignment you will use these programs to test your compiler, so the tests should be thorough and should

exercise all the features of P_0. If the tests are not thorough, then your compiler may pass all your tests but still have bugs that are caught by the automatic grader. Run your test programs using the standard Python interpreter.

1.3. Write recursive functions

The main programming technique for analyzing and manipulating ASTs is to write recursive functions that traverse the tree. As an example, we create a function called `num_nodes` that counts the number of nodes in an AST. Figure 3 shows a schematic of how this function works. Each triangle represents a call to `num_nodes` and is responsible for counting the number of nodes in the sub-tree whose root is the argument to `num_nodes`. In the figure, the largest triangle is responsible for counting the number of nodes in the sub-tree rooted at `Add`. The key to writing a recursive function is to be lazy! Let the recursion do the work for you. Just process one node and let the recursion handle the children. In Figure 3 we make the recursive calls `num_nodes(left)` and `num_nodes(right)` to count the nodes in the child sub-trees. All we have to do to then is to add the two numbers and add one more to count the current node. Figure 4 shows the definition of the `num_nodes` function.

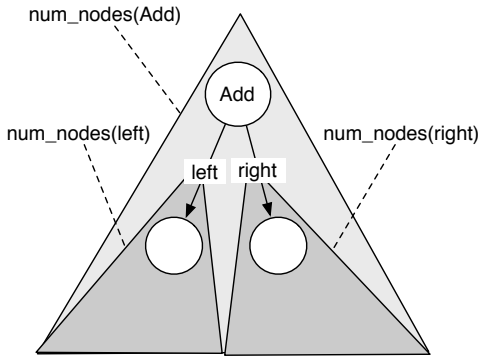


FIGURE 3. Schematic for a recursive function processing an AST.

When a node has a *list* of children, as is the case for `Stmt`, a convenient way to process the children is to use List Comprehensions, described in the Python Tutorial [**21**]. A list comprehension has the following form

[*compute* `for` *variable* `in` *list*]

```
from compiler.ast import *

def num_nodes(n):
    if isinstance(n, Module):
        return 1 + num_nodes(n.node)
    elif isinstance(n, Stmt):
        return 1 + sum([num_nodes(x) for x in n.nodes])
    elif isinstance(n, Printnl):
        return 1 + num_nodes(n.nodes[0])
    elif isinstance(n, Assign):
        return 1 + num_nodes(n.nodes[0]) + num_nodes(n.expr)
    elif isinstance(n, AssName):
        return 1
    elif isinstance(n, Discard):
        return 1 + num_nodes(n.expr)
    elif isinstance(n, Const):
        return 1
    elif isinstance(n, Name):
        return 1
    elif isinstance(n, Add):
        return 1 + num_nodes(n.left) + num_nodes(n.right)
    elif isinstance(n, UnarySub):
        return 1 + num_nodes(n.expr)
    elif isinstance(n, CallFunc):
        return 1 + num_nodes(n.node)
    else:
        raise Exception('Error in num_nodes: unrecognized AST node')
```

FIGURE 4. Recursive function that counts the number of nodes in an AST.

This performs the specified computation for each element in the list, resulting in a list holding the results of the computations. For example, in Figure 4 in the case for `Stmt` we write

```
[num_nodes(x) for x in n.nodes]
```

This makes a recursive call to `num_nodes` for each child node in the list `n.nodes`. The result of this list comprehension is a list of numbers. The complete code for handling a `Stmt` node in Figure 4 is

```
return 1 + sum([num_nodes(x) for x in n.nodes])
```

As is typical in a recursive function, after making the recursive calls to the children, there is some work left to do. We add up the number of nodes from the children using the `sum` function, which is documented under Built-in Functions in the Python Library Manual [**19**]. We then add `1` to account for the `Stmt` node itself.

There are 11 `if` statements in the `num_nodes` function, one for each kind of AST node. In general, when writing a recursive function

over an AST, it is good to double check and make sure that you have written one `if` for each kind of AST node. The `raise` of an exception in the `else` checks that the input does not contain any other kinds of nodes.

1.4. Learn the x86 assembly language

This section gives a brief introduction to the x86 assembly language. There are two variations on the syntax for x86 assembly: the Intel syntax and the AT&T syntax. Here we use the AT&T syntax, which is accepted by the GNU Assembler and by `gcc`. The main difference between the AT&T syntax and the Intel syntax is that in AT&T syntax the destination register on the right, whereas in Intel syntax it is on the left.

The x86 assembly language consists of hundreds of instructions and many intricate details. However, for our purposes a tiny subset of the language will do. The program in Figure 5 serves to give a first taste of x86 assembly. This program is equivalent to the following Python program, a small variation on the one we discussed earlier.

```
x = - input()
print x + input()
```

Perhaps the most obvious difference between Python and x86 is that Python allows expressions to be nested within one another. In contrast, an x86 program consists of a flat sequence of instructions. Another difference is that x86 does not have variables. Instead, it has a fixed set of ***registers*** that can each hold 32 bits. The registers have funny three letter names:

```
eax, ebx, ecx, edx, esi, edi, ebp, esp
```

When referring to a register in an instruction, place a percent sign (%) before the name of the register.

When compiling from Python to x86, we may very well have more variables than registers. In such situations we use the ***stack*** to store the variables. Recall that a stack is a data structure that supports pushing and popping values in a first-in-first-out (FIFO) manner. In the program in Figure 5, the variable `x` has been mapped to a stack location. The register `esp` always contains the address of the item at the front of the stack. In addition to local variables, the stack is also used to pass arguments in a function call. In this course we use the cdecl convention used by the GNU C compiler. The instruction `pushl %eax`, which appears before the call to `print_int_nl`, serves to put the result of the addition on the stack so that it can be

accessed within the `print_int_nl` function. The stack grows down, so the `pushl` instruction causes `esp` to be lowered by 4 bytes (the size of one 32 bit integer).

Because the stack pointer, `esp`, is constantly changing, it would be difficult to use `esp` for referring to local variables stored on the stack. Instead, the `ebp` register (`bp` is for base pointer) is used for this purpose.

The stack is conceptually a two-dimensional stack. We have already seen that words (32-bit values) are pushed and popped via the stack pointer. Additionally, each function call pushes an *activation record* to store its local state. The function call's activation record is popped when the function returns. The first two instructions set up of the activation record. In particular, the instruction `pushl %ebp` saves the current value of the base pointer, that is, the base pointer of the previous activation record. Then, the instruction `movl %esp,%ebp` puts a copy of the stack pointer into `ebp` delineating the start of this call's activation record. We can then use `ebp` throughout the lifetime of the call to this function to refer to local variables on the stack. And we see that `ebp` points to the head of a linked-list of activation records. In Figure 5, the value of variable `x` is referred to by `-4(%ebp)`, which is the assembly way of writing the C expression `*(ebp - 4)` (the `4` is in bytes). That is, it loads the data from the address stored in `ebp` minus 4 bytes.

The `eax` register plays a special role: functions put their return values in `eax`. For example, in Figure 5, the calls to the `input` function put their results in `eax`. It is a good idea not to put anything important into `eax` before making a function call, as it will be overwritten. In general, a function may overwrite any of the *caller-save* registers, which are `eax`, `ecx`, and `edx`. The rest of the registers are *callee-save* which means that if a function wants to use those registers, it has to first save them and then restore them before returning. In Figure 5, the first thing that the `main` function does is save the value of the `ebp` register on the stack. The `leave` instruction copies the contents of the `ebp` register into the `esp` register, so `esp` points to the same place in the stack as the base pointer. It then pops the old base pointer into the `ebp` register. Appendix 6.4 is a quick reference for the x86 instructions that you will likely need. (They are the ones I used in the reference compiler.) For a complete list of x86 instructions, see the Intel manuals [**9, 10, 11**].

```
.globl main
main:
        pushl %ebp
        movl %esp, %ebp
        subl $4, %esp
        call input
        negl %eax
        movl %eax, -4(%ebp)
        call input
        addl -4(%ebp), %eax
        pushl %eax
        call print_int_nl
        addl $4, %esp
        movl $0, %eax
        leave
        ret
```

FIGURE 5. The x86 assembly code for the program `x = - input(); print x + input()`.

1.5. Flatten expressions

The first step in translating from P_0 to x86 is to flatten complex expressions into a series of assignment statements. For example, the program

```
print - input() + 2
```

is translated to the following

```
tmp0 = input()
tmp1 = - tmp0
tmp2 = tmp1 + 2
print tmp2
```

In the resulting code, the operands of an expression are either variables or constants, that is, they are *simple expressions*. If an expression has any other kind of operand, then it is a *complex* expression.

Exercise 1.3. Write a recursive function that flattens a P_0 program into an equivalent P_0 program that contains no complex expressions. Test that you have not changed the semantics of the program by writing a function that prints the resulting program. Run the program using the standard python interpreter to verify that it gives the the same answers for all of your test cases.

1.6. Select instructions

The next step is to translate the flattened P_0 statements into x86 instructions. For now we will assign all variables to locations on the stack. In chapter 3, we describe a register allocation algorithm that tries to place as many variables as possible into registers.

Figure 6 shows an example translation, selecting x86 instructions to accomplish each P_0 statement. Sometimes several x86 instruction are needed to carry out a Python statement. The translation shown here strives for simplicity over performance. You are encouraged to experiment with better instruction sequences, but it is recommended that you only do that after getting a simple version working. The `print_int_nl` function is provided in a C library, the file `runtime.c` on the course web page.

Exercise 1.4. Write a function that translates flattened P_0 programs into x86 assembly. In addition to selecting instructions for the Python statements, you will need to generate a label for the `main` function and the proper prologue and epilogue instructions, which you can copy from Figure 5. The suggested organization of your compiler is shown in Figure 7.

Your compiler should be a Python script that takes one argument, the name of the input file, and that produces a file (containing the x86 ouput) with the same name as the input file except that the `.py` suffix should be replaced by the `.s` suffix.

```
  tmp0 = input()
  tmp1 = - tmp0
  tmp2 = tmp1 + 2
  print tmp2
⟹
.globl main
main:
        pushl %ebp
        movl %esp, %ebp
        subl $12,%esp           # make stack space for variables

        call input
        movl %eax, -4(%ebp)     # tmp0 is in 4(%ebp)

        movl -4(%ebp), %eax
        movl %eax, -8(%ebp)     # tmp1 is in 8(%ebp)
        negl -8(%ebp)

        movl -8(%ebp), %eax,
        movl %eax, -12(%ebp)    # tmp2 is in 12(%ebp)
        addl $2, -12(%ebp)

        pushl -12(%ebp)         # push the argument on the stack
        call print_int_nl
        addl $4, %esp           # pop the stack

        movl $0, %eax           # put return value in eax
        leave
        ret
```

FIGURE 6. Example translation to x86 assembly.

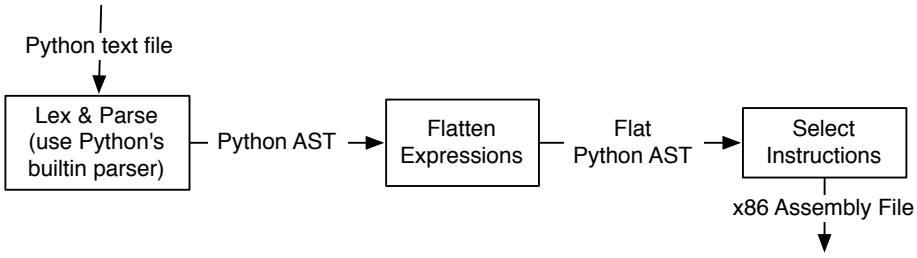


FIGURE 7. Suggested organization of the compiler.

CHAPTER 2

Parsing

The main ideas covered in this chapter are

lexical analysis: the identification of tokens (i.e., words) within sequences of characters.

parsing: the identification of sentence structure within sequences of tokens.

In general, the syntax of the source code for a language is called its *concrete syntax*. The concrete syntax of P_0 specifies which programs, expressed as sequences of characters, are P_0 programs. The process of transforming a program written in the concrete syntax (a sequence of characters) into an abstract syntax tree is traditionally subdivided into two parts: *lexical analysis* (often called scanning) and *parsing*. The lexical analysis phase translates the sequence of characters into a sequence of *tokens*, where each token consists of several characters. The parsing phase organizes the tokens into a *parse tree* as directed by the grammar of the language and then translates the parse tree into an abstract syntax tree.

It is feasible to implement a compiler without doing lexical analysis, instead just parsing. However, scannerless parsers tend to be slower, which mattered back when computers were slow, and sometimes still matters for very large files.

The Python Lex-Yacc tool, abbreviated PLY [**2**], is an easy-to-use Python imitation of the original `lex` and `yacc` C programs. Lex was written by Eric Schmidt and Mike Lesk [**14**] at Bell Labs, and is the standard lexical analyzer generator on many Unix systems. YACC stands from Yet Another Compiler Compiler and was originally written by Stephen C. Johnson at AT&T [**12**]. The PLY tool combines the functionality of both `lex` and `yacc`. In this chapter we will use the PLY tool to generate a lexer and parser for the P_0 subset of Python.

2.1. Lexical analysis

The lexical analyzer turns a sequence of characters (a string) into a sequence of tokens. For example, the string

```
'print 1 + 3'
```

will be converted into the list of tokens

```
['print','1','+','3']
```

Actually, to be more accurate, each token will contain the token `type` and the token's `value`, which is the string from the input that matched the token.

With the PLY tool, the types of the tokens must be specified by initializing the `tokens` variable. For example,

```
tokens = ('PRINT','INT','PLUS')
```

To construct the lexical analyzer, we must specify which sequences of characters will map to each type of token. We do this specification using regular expressions. The term "regular" comes from "regular languages", which are the (particularly simple) class of languages that can be recognized by a finite automaton. A "language" is a set of strings. A *regular expression* is a pattern formed of the following core elements:

(1) a character, e.g. `a`. The only string that matches this regular expression is `'a'`.
(2) two regular expressions, one followed by the other (concatenation), e.g. `bc`. The only string that matches this regular expression is `'bc'`.
(3) one regular expression or another (alternation), e.g. `a|bc`. Both the string `'a'` and `'bc'` would be matched by this pattern (i.e., the language described by the regular expression `a|bc` consists of the strings `'a'` and `'bc'`).
(4) a regular expression repeated zero or more times (Kleene closure), e.g. `(a|bc)*`. The string `'bcabcbc'` would match this pattern, but not `'bccba'`.
(5) the empty sequence (epsilon)

The Python support for regular expressions goes beyond the core elements and includes many other convenient short-hands, for example `+` is for repetition one or more times. If you want to refer to the actual character `+`, use a backslash to escape it. Section 4.2.1 Regular Expression Syntax of the Python Library Reference gives an in-depth description of the extended regular expressions supported by Python.

Normal Python strings give a special interpretation to backslashes, which can interfere with their interpretation as regular expressions. To avoid this problem, use Python's raw strings instead of normal

strings by prefixing the string with an `r`. For example, the following specifies the regular expression for the `'PLUS'` token.

```
t_PLUS =   r'\+'
```

The `t_` is a naming convention that PLY uses to know when you are defining the regular expression for a token.

Sometimes you need to do some extra processing for certain kinds of tokens. For example, for the `INT` token it is nice to convert the matched input string into a Python integer. With PLY you can do this by defining a function for the token. The function must have the regular expression as its documentation string and the body of the function should overwrite in the `value` field of the token. Here's how it would look for the `INT` token. The `\d` regular expression stands for any decimal numeral (0-9).

```
def t_INT(t):
    r'\d+'
    try:
      t.value = int(t.value)
    except ValueError:
      print "integer value too large", t.value
      t.value = 0
    return t
```

In addition to defining regular expressions for each of the tokens, you'll often want to perform special handling of newlines and whitespace. The following is the code for counting newlines and for telling the lexer to ignore whitespace. (Python has complex rules for dealing with whitespace that we'll ignore for now.)

```
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

t_ignore  = ' \t'
```

If a portion of the input string is not matched by any of the tokens, then the lexer calls the error function that you provide. The following is an example error function.

```
def t_error(t):
    print "Illegal character '%s'" % t.value[0]
    t.lexer.skip(1)
```

Last but not least, you'll need to instruct PLY to generate the lexer from your specification with the following code.

```
import ply.lex as lex
lex.lex()
```

Figure 1 shows the complete code for an example lexer.

```
tokens = ('PRINT','INT','PLUS')

t_PRINT = r'print'

t_PLUS =   r'\+'

def t_INT(t):
    r'\d+'
    try:
      t.value = int(t.value)
    except ValueError:
      print "integer value too large", t.value
      t.value = 0
    return t

t_ignore  = ' \t'

def t_newline(t):
  r'\n+'
  t.lexer.lineno += t.value.count("\n")

def t_error(t):
  print "Illegal character '%s'" % t.value[0]
  t.lexer.skip(1)

import ply.lex as lex
lex.lex()
```

FIGURE 1. Example lexer implemented using the PLY lexer generator.

Exercise 2.1. Write a PLY lexer specification for P_0 and test it on a few input programs, looking at the output list of tokens to see if they make sense.

2.2. Background on CFGs and the P_0 grammar.

A *context-free grammar* (CFG) consists of a set of *rules* (also called productions) that describes how to categorize strings of various forms.

Context-free grammars specify a class of languages known as *context-free languages* (like regular expressions specify regular languages). There are two kinds of categories, *terminals* and *non-terminals* in a context-free grammar. The terminals correspond to the tokens from the lexical analysis. Non-terminals are used to categorize different parts of a language, such as the distinction between statements and expressions in Python and C. The term *symbol* refers to both terminals and non-terminals. A grammar rule has two parts, the left-hand side is a non-terminal and the right-hand side is a sequence of zero or more symbols. The notation `::=` is used to separate the left-hand side from the right-hand side. The following is a rule that could be used to specify the syntax for an addition operator.

(1) `expression ::= expression PLUS expression`

This rule says that if a string can be divided into three parts, where the first part can be categorized as an expression, the second part is the `PLUS` terminal (token), and the third part can be categorized as an expression, then the entire string can be categorized as an expression. The next example rule has the terminal `INT` on the right-hand side and says that a string that is categorized as an integer (by the lexer) can also be categorized as an expression. As is apparent here, a string can be categorized by more than one non-terminal.

(2) `expression ::= INT`

To *parse* a string is to determine how the string can be categorized according to a given grammar. Suppose we have the string "`1 + 3`". Both the `1` and the `3` can be categorized as expressions using rule 2. We can then use rule 1 to categorize the entire string as an expression. A *parse tree* is a good way to visualize the parsing process. (You will be tempted to confuse parse trees and abstract syntax trees. There is a close correspondence, but the excellent students will carefully study the difference to avoid this confusion.) A parse tree for "`1 + 3`" is shown in Figure 2. The best way to start drawing a parse tree is to first list the tokenized string at the bottom of the page. These tokens correspond to terminals and will form the leaves of the parse tree. You can then start to categorize non-terminals, or sequences of non-terminals, using the parsing rules. For example, we can categorize the integer "`1`" as an expression using rule (2), so we create a new node above "`1`", label the node with the left-hand side terminal, in this case `expression`, and draw a line down from the new node down to "`1`". As an optional step, we can record which rule we used in parenthesis after the name of the terminal. We then

repeat this process until all of the leaves have been connected into a single tree, or until no more rules apply.

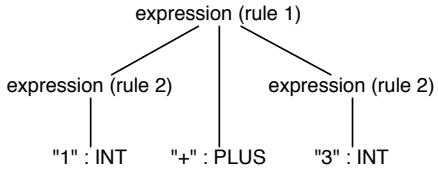


FIGURE 2. The parse tree for "`1 + 3`".

Exhibiting a parse tree for a string validates that it is in the language described by the context-free grammar in question. If there can be more than one parse tree for the same string, then the grammar is *ambiguous*. For example, the string "`1 + 2 + 3`" can be parsed two different ways using rules 1 and 2, as shown in Figure 3. In Section 2.4.2 we'll discuss ways to avoid ambiguity through the use of precedence levels and associativity.

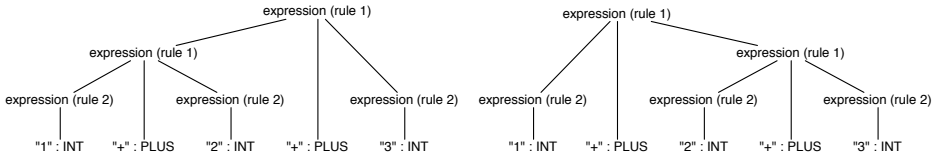


FIGURE 3. Two parse trees for "`1 + 2 + 3`".

The process described above for creating a parse-tree was "bottom-up". We started at the leaves of the tree and then worked back up to the root. An alternative way to build parse-trees is the "top-down" *derivation* approach. This approach is not a practical way to parse a particular string but it is helpful for thinking about all possible strings that are in the language described by the grammar. To perform a derivation, start by drawing a single node labeled with the starting non-terminal for the grammar. This is often the `program` non-terminal, but in our case we simply have `expression`. We then select at random any grammar rule that has `expression` on the left-hand side and add new edges and nodes to the tree according to the right-hand side of the rule. The derivation process then repeats by selecting another non-terminal that does not yet have children. Figure 4 shows the process of building a parse tree by derivation. A *left-most derivation* is one in which the left-most non-terminal is

always chosen as the next non-terminal to expand. A `right-most derivation` is one in which the right-most non-terminal is always chosen as the next non-terminal to expand. The derivation in Figure 4 is a right-most derivation.

FIGURE 4. Building a parse-tree by derivation.

For each subset of Python in this course, we will specify which language features are in a given subset of Python using context-free grammars. The notation we'll use for grammars is Extended Backus-Naur Form (EBNF). The grammar for P_0 is shown in Figure 5. Any symbol not appearing on the left-hand side of a rule is a terminal (e.g., `name` and `decimalinteger`). For simple terminals consisting of single strings, we simply use the string and avoid giving names to them (e.g., `"+"`). This notation does not correspond exactly to the notation for grammars used by PLY, but it should not be too difficult for the reader to figure out the PLY grammar given the EBNF grammar.

```
program ::= module
module ::= simple_statement+
simple_statement ::= "print" expression
                   | name "=" expression
                   | expression
expression ::= name
             | decimalinteger
             | "-" expression
             | expression "+" expression
             | "(" expression ")"
             | "input" "(" ")"
```

FIGURE 5. Context-free grammar for the P_0 subset of Python.

2.3. Generating parsers with PLY

Figure 6 shows an example use of PLY to generate a parser. The code specifies a grammar and it specifies actions for each rule. For each grammar rule there is a function whose name must begin with `p_`. The document string of the function contains the specification of the grammar rule. PLY uses just a colon `:` instead of the usual `::=`

to separate the left and right-hand sides of a grammar production. The left-hand side symbol for the first function (as it appears in the Python file) is considered the start symbol. The body of these functions contains code that carries out the action for the production.

Typically, what you want to do in the actions is build an abstract syntax tree, as we do here. The parameter `t` of the function contains the results from the actions that were carried out to parse the right-hand side of the production. You can index into `t` to access these results, starting with `t[1]` for the first symbol of the right-hand side. To specify the result of the current action, assign the result into `t[0]`. So, for example, in the production `expression : INT`, we build a `Const` node containing an integer that we obtain from `t[1]`, and we assign the `Const` node to `t[0]`.

```
from compiler.ast import Printnl, Add, Const

def p_print_statement(t):
  'statement : PRINT expression'
  t[0] = Printnl([t[2]], None)

def p_plus_expression(t):
  'expression : expression PLUS expression'
  t[0] = Add((t[1], t[3]))

def p_int_expression(t):
  'expression : INT'
  t[0] = Const(t[1])

def p_error(t):
  print "Syntax error at '%s'" % t.value

import ply.yacc as yacc
yacc.yacc()
```

FIGURE 6. First attempt at writing a parser using PLY.

The PLY parser generator takes your grammar and generates a parser that uses the LALR(1) shift-reduce algorithm, which is the most common parsing algorithm in use today. LALR(1) stands for Look Ahead Left-to-right with Rightmost-derivation and 1 token of lookahead. Unfortunately, the LALR(1) algorithm cannot handle all context-free grammars, so sometimes you will get error messages from PLY. To understand these errors and know how to avoid them, you have to know a little bit about the parsing algorithm.

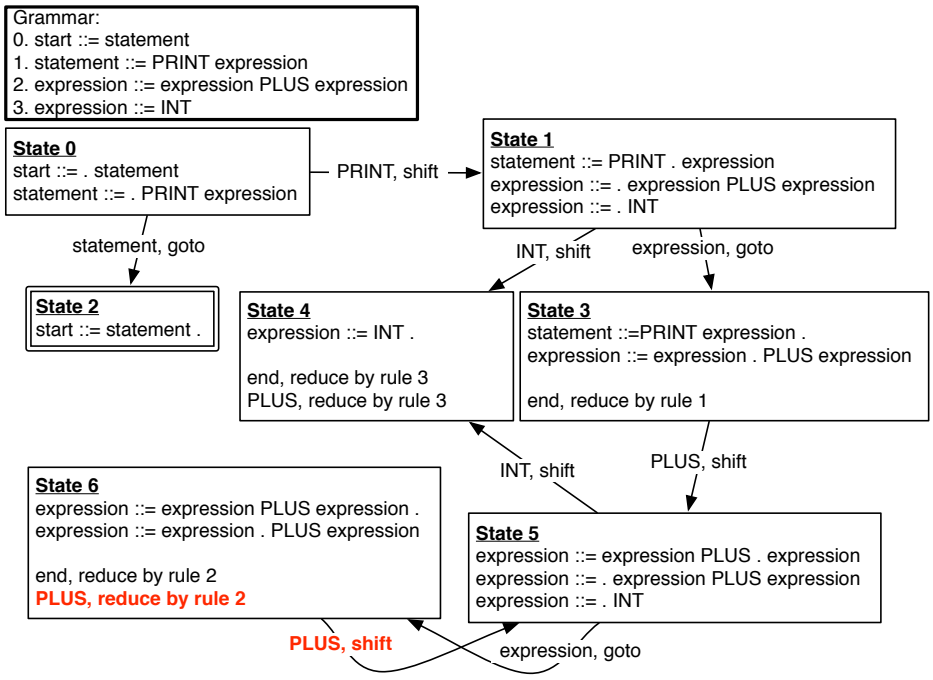
2.4. The LALR(1) algorithm

To understand the error messages of PLY, one needs to understand the underlying parsing algorithm. The LALR(1) algorithm uses a stack and a finite automaton. Each element of the stack is a pair: a state number and a symbol. The symbol characterizes the input that has been parsed so-far and the state number is used to remember how to proceed once the next symbol-worth of input has been parsed. Each state in the finite automaton represents where the parser stands in the parsing process with respect to certain grammar rules. Figure 7 shows an example LALR(1) parse table generated by PLY for the grammar specified in Figure 6. When PLY generates a parse table, it also outputs a textual representation of the parse table to the file `parser.out` which is useful for debugging purposes.

Consider state 1 in Figure 7. The parser has just read in a `PRINT` token, so the top of the stack is `(1,PRINT)`. The parser is part of the way through parsing the input according to grammar rule 1, which is signified by showing rule 1 with a dot after the PRINT token and before the expression non-terminal. A rule with a dot in it is called an *item*. There are several rules that could apply next, both rule 2 and 3, so state 1 also shows those rules with a dot at the beginning of their right-hand sides. The edges between states indicate which transitions the automaton should make depending on the next input token. So, for example, if the next input token is INT then the parser will push INT and the target state 4 on the stack and transition to state 4. Suppose we are now at the end of the input. In state 4 it says we should reduce by rule 3, so we pop from the stack the same number of items as the number of symbols in the right-hand side of the rule, in this case just one. We then momentarily jump to the state at the top of the stack (state 1) and then follow the goto edge that corresponds to the left-hand side of the rule we just reduced by, in this case `expression`, so we arrive at state 3. (A slightly longer example parse is shown in Figure 7.)

In general, the shift-reduce algorithm works as follows. Look at the next input token.

- If there there is a shift edge for the input token, push the edge's target state and the input token on the stack and proceed to the edge's target state.
- If there is a reduce action for the input token, pop k elements from the stack, where k is the number of symbols in the right-hand side of the rule being reduced. Jump to the state at the top of the stack and then follow the goto edge



Example parse of 'print 1 + 2'

Stack	Input	Action
[]	'print 1 + 2'	shift to state 1
[(1,PRINT)]	'1 + 2'	shift to state 4
[(1,PRINT),(4,INT)]	'+ 2'	reduce by rule 3 to state 1, goto 3
[(1,PRINT),(3,expression)]	'+ 2'	shift to state 5
[(1,PRINT),(3,expression),(5,+)]	'2'	shift to state 4
[(1,PRINT),(3,expression),(5,+),(4,INT)]	''	reduce by rule 3 to state 5, goto 6
[(1,PRINT),(3,expression),(5,+),(6,expression)]	''	reduce by rule 2 to state 1, goto 3
[(1,PRINT),(3,expression)]	''	reduce by rule 1 to state 0, goto 2
[(2,statement)]	''	accept

FIGURE 7. An LALR(1) parse table and a trace of an example run.

for the non-terminal that matches the left-hand side of the rule we're reducing by. Push the edge's target state and the non-terminal on the stack.

Notice that in state 6 of Figure 7 there is both a shift and a reduce action for the token `PLUS`, so the algorithm does not know which action to take in this case. When a state has both a shift and a reduce action for the same token, we say there is a *shift/reduce conflict*. In this case, the conflict will arise, for example, when trying to parse the input `print 1 + 2 + 3`. After having consumed `print 1 + 2` the parser will be in state 6, and it will not know whether to reduce to form

an expression of `1 + 2`, or whether it should proceed by shifting the next + from the input.

A similar kind of problem, known as a *reduce/reduce* conflict, arises when there are two reduce actions in a state for the same token. To understand which grammars gives rise to shift/reduce and reduce/reduce conflicts, it helps to know how the parse table is generated from the grammar, which we discuss next.

2.4.1. Parse table generation. The parse table is generated one state at a time. State 0 represents the start of the parser. We add the production for the start symbol to this state with a dot at the beginning of the right-hand side. If the dot appears immediately before another non-terminal, we add all the productions with that non-terminal on the left-hand side. Again, we place a dot at the beginning of the right-hand side of each the new productions. This process called *state closure* is continued until there are no more productions to add. We then examine each item in the current state I. Suppose an item has the form $A ::= \alpha.X\beta$, where A and X are symbols and α and β are sequences of symbols. We create a new state, call it J. If X is a terminal, we create a shift edge from I to J, whereas if X is a non-terminal, we create a goto edge from I to J. We then need to add some items to state J. We start by adding all items from state I that have the form $B ::= \gamma.X\kappa$ (where B is any symbol and γ and κ are arbitrary sequences of symbols), but with the dot moved past the X. We then perform state closure on J. This process repeats until there are no more states or edges to add.

We then mark states as accepting states if they have an item that is the start production with a dot at the end. Also, to add in the reduce actions, we look for any state containing an item with a dot at the end. Let n be the rule number for this item. We then put a reduce n action into that state for every token Y. For example, in Figure 7 state 4 has an item with a dot at the end. We therefore put a reduce by rule 3 action into state 4 for every token. (Figure 7 does not show a reduce rule for INT in state 4 because this grammar does not allow two consecutive INT tokens in the input. We will not go into how this can be figured out, but in any event it does no harm to have a reduce rule for INT in state 4; it just means the input will be rejected at a later point in the parsing process.)

Exercise 2.2. On a piece of paper, walk through the parse table generation process for the grammar in Figure 6 and check your results against Figure 7.

2.4.2. Resolving conflicts with precedence declarations. To solve the shift/reduce conflict in state 6, we can add the following precedence rule, which says addition associates to the left and takes precedence over printing. This will cause state 6 to choose reduce over shift.

```
precedence = (
    ('nonassoc','PRINT'),
    ('left','PLUS')
    )
```

In general, the precedence variable should be assigned a tuple of tuples. The first element of each inner tuple should be an associativity (nonassoc, left, or right) and the rest of the elements should be tokens. The tokens that appear in the same inner tuple have the same precedence, whereas tokens that appear in later tuples have a higher precedence. Thus, for the typical precedence for arithmetic operations, we would specify the following:

```
precedence = (
    ('left','PLUS','MINUS'),
    ('left','TIMES','DIVIDE')
    )
```

Figure 8 shows the Python code for generating a lexer and parser using PLY.

Exercise 2.3. Write a PLY grammar specification for P_0 and update your compiler so that it uses the generated lexer and parser instead of using the parser in the `compiler` module. In addition to handling the grammar in Figure 5, you also need to handle Python-style comments, everything following a `#` symbol up to the newline should be ignored. Perform regression testing on your compiler to make sure that it still passes all of the tests that you created for P_0.

```
# Lexer
tokens = ('PRINT','INT','PLUS')
t_PRINT = r'print'
t_PLUS = r'\+'
def t_INT(t):
    r'\d+'
    try:
        t.value = int(t.value)
    except ValueError:
        print "integer value too large", t.value
        t.value = 0
    return t
t_ignore  = ' \t'
def t_newline(t):
    r'\n+'
    t.lexer.lineno += t.value.count("\n")
def t_error(t):
    print "Illegal character '%s'" % t.value[0]
    t.lexer.skip(1)
import ply.lex as lex
lex.lex()
# Parser
from compiler.ast import Printnl, Add, Const
precedence = (
    ('nonassoc','PRINT'),
    ('left','PLUS')
    )
def p_print_statement(t):
    'statement : PRINT expression'
    t[0] = Printnl([t[2]], None)
def p_plus_expression(t):
    'expression : expression PLUS expression'
    t[0] = Add((t[1], t[3]))
def p_int_expression(t):
    'expression : INT'
    t[0] = Const(t[1])
def p_error(t):
    print "Syntax error at '%s'" % t.value
import ply.yacc as yacc
yacc.yacc()
```

FIGURE 8. Example parser with precedence declarations to resolve conflicts.

CHAPTER 3

Register allocation

In chapter 1 we simplified the generation of x86 assembly by placing all variables on the stack. We can improve the performance of the generated code considerably if we instead try to place as many variables as possible into registers. The CPU can access a register in a single cycle, whereas accessing the stack can take from several cycles (to go to cache) to hundreds of cycles (to go to main memory). Figure 1 shows a program fragment that we'll use as a running example. The program is almost in x86 assembly but not quite; it still contains variables instead of stack locations or registers.

The goal of register allocation is to fit as many variables into registers as possible. It is often the case that we have more variables than registers, so we can't naively map each variable to a register. Fortunately, it is also common for different variables to be needed during different periods of time, and in such cases the variables can be mapped to the same register. Consider variables `y` and `z` in Figure 1. After the variable `z` is used in `addl z, x` it is no longer needed. Variable `y`, on the other hand, is only used after this point, so `z` and `y` could share the same register.

```
movl $4, z
movl $0, w
movl $1, z
movl w, x
addl z, x
movl w, y
addl x, y
movl y, w
addl x, w
```

FIGURE 1. An example program in pseudo assembly code. The program still uses variables instead of registers and stack locations.

3.1. Liveness analysis

A variable whose current value is needed later on in the program is called *live*.

Definition 3.1. A variable is ***live*** if the variable is used at some later point in the program and there is not an intervening assignment to the variable.

To understand the latter condition, consider variable `z` in Figure 1. It is not live immediately after the instruction `movl $4, z` because the later uses of `z` get their value instead from the instruction `movl $1, z`. The variable `z` is live between `z = 1` and its use in `addl z, x`. We have annotated the program with the set of variables that are live between each instruction.

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

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

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

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

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

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

where $W(k)$ are the variables written to by instruction I_k and $R(k)$ are the variables read by instruction I_k. Figure 2 shows the results of live variables analysis for the example program from Figure 1.

Implementing the live variable analysis in Python is straightforward thanks to the built-in support for sets. You can construct a set by first creating a list and then passing it to the `set` function. The following creates an empty set:

```
>>> set([])
set([])
```

You can take the union of two sets with the `|` operator:

```
>>> set([1,2,3]) | set([3,4,5])
set([1, 2, 3, 4, 5])
```

To take the difference of two sets, use the `-` operator:

```
movl $4, z
                {}
movl $0, w
                {w}
movl $1, z
                {w, z}
movl w, x
                {x, w, z}
addl z, x
                {x, w}
movl w, y
                {x, y}
addl x, y
                {x, y}
movl y, w
                {w, x}
addl x, w
                {}
```

FIGURE 2. The example program annotated with the set of live variables between each instruction.

```
>>> set([1,2,3]) - set([3,4,5])
set([1, 2])
```

Also, just like lists, you can use Python's `for` loop to iterate through the elements of a set:

```
>>> for x in set([1,2,2,3]):
...   print x
1
2
3
```

3.2. Building the interference graph

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

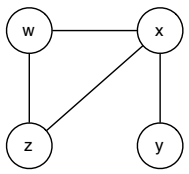


FIGURE 3. Interference graph for the example program.

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

A better way to compute the edges of the intereference graph is given by the following rules.

- If instruction I_k is a move: `movl` s, t (and $t \in L_{\mathsf{after}}(k)$), then add the edge (t, v) for every $v \in L_{\mathsf{after}}(k)$ unless $v = t$ or $v = s$.
- If instruction I_k is not a move but some other arithmetic instruction such as `addl` s, t (and $t \in L_{\mathsf{after}}(k)$), then add the edge (t, v) for every $v \in L_{\mathsf{after}}(k)$ unless $v = t$.
- If instruction I_k is of the form `call` $label$, then add an edge (r, v) for every caller-save register r and every variable $v \in L_{\mathsf{after}}(k)$. (The caller-save registers are `eax`, `ecx`, and `edx`.)

Working from the top to bottom of Figure 2, `z` interferes with `w` and `x`, `w` interferes with `x`, and `y` interferes with `x`. In the second to last statement, we see that `w` interferes with `x`, but we already know that. The resulting interference graph is shown in Figure 3.

In Python, a convenient representation for graphs is to use a dictionary that maps nodes to a set of adjacent nodes. So for the interference graph, the dictionary would map variable names to sets of variable names.

3.3. Color the interference graph by playing Sudoku

We now come to the main event, mapping variables to registers (or to stack locations in the event that we run out of registers). We need to make sure not to map two variables to the same register if the two variables interfere with each other. In terms of the interference

graph, this means we cannot map adjacent nodes to the same register. If we think of registers as colors, the register allocation problem becomes the widely-studied graph coloring problem [**1, 17**].

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

- There is one node in the graph for each Sudoku square.
- There is an edge between two nodes if the corresponding squares are in the same row or 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 nodes in the graph.

If you can color the remaining nodes in the graph with the nine colors, then you've also solved the corresponding game of Sudoku.

Given that Sudoku is graph coloring, one can use Sudoku strategies to come up with an algorithm for allocating registers. For example, one of the basic techniques for Sudoku is Pencil Marks. The idea is that you use a process of elimination to determine what numbers still make sense for a square, and write down those numbers in the square (writing very small). At first, each number might be a possibility, but as the board fills up, more and more of the possibilities are crossed off (or erased). For example, if the number 1 is assigned to a square, then by process of elimination, you can cross off the 1 pencil mark from all the squares in the same row, column, and region. Many Sudoku computer games provide automatic support for Pencil Marks. This heuristic also reduces the degree of branching in the search tree.

The Pencil Marks technique corresponds to the notion of color *saturation* due to Brélaz [**3**]. The saturation of a node, in Sudoku terms, is the number of possibilities that have been crossed off using the process of elimination mentioned above. In graph terminology, we have the following definition:

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

where $\text{Adj}(u)$ is the set of nodes adjacent to u and the notation $|S|$ stands for the size of the set S.

Algorithm: DSATUR
Input: the inference graph G
Output: an assignment $\mathrm{color}(v)$ for each node $v \in G$

$W \leftarrow \mathit{vertices}(G)$
while $W \neq \emptyset$ do
  pick a node u from W with the highest saturation,
    breaking ties randomly
  find the lowest color c that is not in $\{\mathrm{color}(v) \mid v \in \mathrm{Adj}(v)\}$
  $\mathrm{color}(u) = c$
  $W \leftarrow W - \{u\}$

FIGURE 4. Saturation-based greedy graph coloring algorithm.

Using the Pencil Marks technique leads to a simple strategy for filling in numbers: if there is a square with only one possible number left, then write down that number! But what if there aren't any squares with only one possibility left? One brute-force approach is to just make a guess. If that guess ultimately leads to a solution, great. If not, backtrack to the guess and make a different guess. Of course, this is horribly time consuming. One standard way to reduce the amount of backtracking is to use the most-constrained-first heuristic. That is, when making a guess, always choose a square with the fewest possibilities left (the node with the highest saturation). The idea is that choosing highly constrained squares earlier rather than later is better because later there may not be any possibilities left.

In some sense, register allocation is easier than Sudoku because we can always cheat and add more numbers by spilling variables to the stack. Also, we'd like to minimize the time needed to color the graph, and backtracking is expensive. Thus, it makes sense to keep the most-constrained-first heuristic but drop the backtracking in favor of greedy search (guess and just keep going). Figure 4 gives the pseudo-code for this simple greedy algorithm for register allocation based on saturation and the most-constrained-first heuristic, which is roughly equivalent to the DSATUR algorithm of Brélaz [**3**] (also known as saturation degree ordering (SDO) [**7, 15**]). Just as in Sudoku, the algorithm represents colors with integers, with the first k colors corresponding to the k registers in a given machine and the rest of the integers corresponding to stack locations.

3.4. Generate spill code

In this pass we need to adjust the program to take into account our decisions regarding the locations of the local variables. Recall that x86 assembly only allows one operand per instruction to be a memory access. For instance, suppose we have a move `movl y, x` where `x` and `y` are assigned to different memory locations on the stack. We need to replace this instruction with two instructions, one that moves the contents of `y` into a register and then another instruction that moves the register's contents into `x`. But what register? We could reserve a register for this purpose, and use the same register for every move between two stack locations. However, that would decrease the number of registers available for other uses, sometimes requiring the allocator to spill more variables.

Instead, we recommend creating a new temporary variable (not yet assigned to a register or stack location) and rerunning the register allocator on the new program, where `movl y, x` is replaced by

```
movl y, tmp0
movl tmp0, x
```

The `tmp0` variable will have a very short live range, so it does not make the overall graph coloring problem harder to solve. However, to prevent `tmp0` from being spilled and then needing yet another temporary, we recommend marking `tmp0` as "unspillable" and changing the graph coloring algorithm with respect to how it picks the next node. Instead of breaking ties randomly between nodes with equal saturation, give preference to nodes marked as unspillable.

If you did not need to introduce any new temporaries, then register allocation is complete. Otherwise, you need to go back and do another iteration of live variable analysis, graph building, graph coloring, and generating spill code. When you start the next iteration, do not start from scratch; keep the spill decisions, that is, which variables are spilled and their assigned stack locations, but redo the allocation for the new temporaries and the variables that were assigned to registers.

3.5. Assign homes and remove trivial moves

Once the register allocation algorithm has settled on a coloring, update the program by replacing variables with their homes: registers or stack locations. In addition, delete trivial moves. That is, wherever you have a move between two variables, such as

`movl` y, x

where x and y have been assigned to the same location (register or stack location), delete the move instruction.

Exercise 3.1. Update your compiler to perform register allocation. Test your updated compiler on your suite of test cases to make sure you haven't introduced bugs. The suggested organization of your compiler is shown in Figure 7. What is the time complexity of your register allocation algorithm? If it is greater than $O(n \log n)$, find a way to make it $O(n \log n)$, where n is the number of variables in the program.

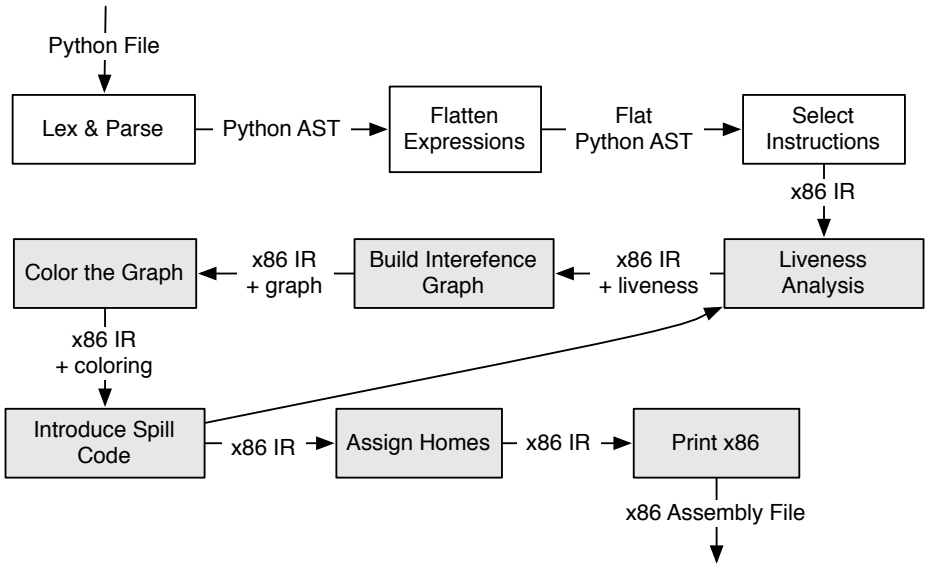


FIGURE 5. Suggested organization of the compiler.

3.6. Read more about register allocation

The general graph coloring problem is NP-complete **[6]**, so finding an optimal coloring (fewest colors) takes exponential time (for example, by using a backtracking algorithm). However, there are many algorithms for finding good colorings and the search for even better algorithms is an ongoing area of research. The most widely used coloring algorithm for register allocation is the classic algorithm of Chaitin **[5]**. Briggs describes several improvements to the classic algorithm **[4]**. Numerous others have also made refinements

and proposed alternative algorithms. The interested reader can google "register allocation".

More recently, researchers have noticed that the interference graphs that arise in compilers using static single-assignment form have a special property, they are chordal graphs. This property allows a simple algorithm to find optimal colorings **[8]**. Furthermore, even if the compiler does not use static single-assignment form, many interference graphs are either chordal or nearly chordal **[16]**.

The chordal graph coloring algorithm consists of putting two standard algorithms together. The first algorithm orders the nodes so that that the next node in the sequence is always the node that is adjacent to the most nodes further back in the sequence. This algorithm is called the maximum cardinality search algorithm (MCS) **[18]**. The second algorithm is the greedy coloring algorithm, which simply goes through the sequence of nodes produced by MCS and assigns a color to each node. The ordering produced by the MCS is similar to the most-constrained-first heuristic: if you've already colored many of the neighbors of a node, then that node likely does not have many possibilities left. The saturation based algorithm presented in Section 3.3 takes this idea a bit further, basing the choice of the next vertex on how many colors have been ruled out for each vertex.

CHAPTER 4

Data types and polymorphism

The main concepts in this chapter are:

polymorphism: dynamic type checking and dynamic dispatch,
control flow: computing different values depending on a conditional expression,
compile time versus run time: the execution of your compiler that performs transformations of the input program versus the execution of the input program after compilation,
type systems: identifying which types of values each expression will produce, and
heap allocation: storing values in memory.

4.1. Syntax of P_1

The P_0 subset of Python only dealt with one kind of data type: plain integers. In this chapter we add Booleans, lists and dictionaries. We also add some operations that work on these new data types, thereby creating the P_1 subset of Python. The syntax for P_1 is shown in Figure 1. We give only the abstract syntax (i.e., assume that all ambiguity is resolved). Any ambiguity is resolved in the same manner as Python. In addition, all of the syntax from P_0 is carried over to P_1 unchanged.

A Python list is a sequence of elements. The standard `python` interpreter uses an array (a contiguous block of memory) to implement a list. A Python dictionary is a mapping from keys to values. The standard `python` interpreter uses a hashtable to implement dictionaries.

4.2. Semantics of P_1

One of the defining characteristics of Python is that it is a dynamically typed language. What this means is that a Python expression may result in many different types of values. For example, the following conditional expression might result in an integer or a list.

```
>>> 2 if input() else [1, 2, 3]
```

```
key_datum ::= expression ":" expression
subscription ::= expression "[" expression "]"
expression ::= "True" | "False"
             | "not" expression
             | expression "and" expression
             | expression "or" expression
             | expression "==" expression
             | expression "!=" expression
             | expression "if" expression "else" expression
             | "[" expr_list "]"
             | "{" key_datum_list "}"
             | subscription
             | expression "is" expression
expr_list ::= ϵ
            | expression
            | expression "," expr_list
key_datum_list ::= ϵ
                 | key_datum
                 | key_datum "," key_datum_list
target ::= identifier
         | subscription
simple_statement ::= target "=" expression
```

FIGURE 1. Syntax for the P_1 subset of Python. (In addition to the syntax of P_0.)

In a statically typed language, such as C++ or Java, the above expression would not be allowed; the type checker disallows expressions such as the above to ensure that each expression can only result in one type of value.

Many of the operators in Python are defined to work on many different types, often performing different actions depending on the run-time type of the arguments. For example, addition of two lists performs concatenation.

```
>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]
```

For the arithmetic operators, `True` is treated as if it were the integer `1` and `False` is treated as `0`. Furthermore, numbers can be used in places where Booleans are expected. The number `0` is treated as `False` and everything else is treated as `True`. Here are a few examples:

```
>>> False + True
```

```
1
>>> False or False
False
>>> 1 and 2
2
>>> 1 or 2
1
```

Note that the result of a logic operation such as `and` and `or` does not necessarily return a Boolean value. Instead, e_1 `and` e_2 evaluates expression e_1 to a value v_1. If v_1 is equivalent to `False`, the result of the `and` is v_1. Otherwise e_2 is evaluated to v_2 and v_2 is the result of the `and`. The `or` operation works in a similar way except that it checks whether v_1 is equivalent to `True`.

A list may be created with an expression that contains a list of its elements surrounded by square brackets, e.g., `[3,1,4,1,5,9]` creates a list of six integers. The nth element of a list can be accessed using the subscript notation `l[n]` where `l` is a list and `n` is an integer (indexing is zero based). For example, `[3,1,4,1,5,9][2]` evaluates to `4`. The nth element of a list can be changed by using a subscript expression on the left-hand side of an assignment. For example, the following fixes the 4th digit of π.

```
>>> x = [3,1,4,8,5,9]
>>> x[3] = 1
>>> print x
[3, 1, 4, 1, 5, 9]
```

A dictionary is created by a set of key-value bindings enclosed in braces. The key and value expression are separated by a colon. You can lookup the value for a key using the bracket, such as `d[7]` below. To assign a new value to an existing key, or to add a new key-value binding, use the bracket on the left of an assignment.

```
>>> d = {42: [3,1,4,1,5,9], 7: True}
>>> d[7]
True
>>> d[42]
[3, 1, 4, 1, 5, 9]
>>> d[7] = False
>>> d
{42: [3, 1, 4, 1, 5, 9], 7: False}
>>> d[0] = 1
>>> d[0]
1
```

With the introduction of lists and dictionaries, we have entities in the language where there is a distinction between identity (the `is` operator) and equality (the == operator). The following program, we create two lists with the same elements. Changing list `x` does not affect list `y`.

```
>>> x = [1,2]
>>> y = [1,2]
>>> print x == y
True
>>> print x is y
False
>>> x[0] = 3
>>> print x
[3, 2]
>>> print y
[1, 2]
```

Variable assignment is shallow in that it just points the variable to a new entity and does not affect the entity previous referred to by the variable. Multiple variables can point to the same entity, which is called *aliasing*.

```
>>> x = [1,2,3]
>>> y = x
>>> x = [4,5,6]
>>> print y
[1, 2, 3]
>>> y = x
>>> x[0] = 7
>>> print y
[7, 5, 6]
```

Exercise 4.1. Read the sections of the Python Reference Manual that apply to P_1: 3.1, 3.2, 5.2.2, 5.2.4, 5.2.6, 5.3.2, 5.9, and 5.10.

Exercise 4.2. Write at least ten programs in the P_1 subset of Python that help you understand the language. Look for corner cases or unusual aspects of the language to test in your programs.

4.3. New Python AST classes

Figure 2 shows the additional Python classes used to represent the AST nodes of P_1. Python represents `True` and `False` as variables (using the `Name` AST class) with names `'True'` and `'False'`. Python allows these names to be assigned to, but for P_1, you may assume that they cannot written to (i.e., like `input`). The `Compare` class is

for representing comparisons such as == and !=. The expr attribute of Compare is for the first argument and the ops member contains a list of pairs, where the first item of each pair is a string specifying the operation, such as '==', and the second item is the argument. For P_1 we are guaranteed that this list only contains a single pair. The And and Or classes each contain a list of arguments, held in the nodes attribute and for P_1 this list is guaranteed to have length 2. The Subscript node represents accesses to both lists and dictionaries and can appear within an expression or on the left-hand-side of an assignment. The flags attribute should be ignored for the time being.

```
class Compare(Node):
    def __init__(self, expr, ops):
        self.expr = expr
        self.ops = ops
class Or(Node):
    def __init__(self, nodes):
        self.nodes = nodes
class And(Node):
    def __init__(self, nodes):
        self.nodes = nodes
class Not(Node):
    def __init__(self, expr):
        self.expr = expr
```

```
class List(Node):
    def __init__(self, nodes):
        self.nodes = nodes
class Dict(Node):
    def __init__(self, items):
        self.items = items
class Subscript(Node):
    def __init__(self, expr, flags, subs):
        self.expr = expr
        self.flags = flags
        self.subs = subs
class IfExp(Node):
    def __init__(self, test, then, else_):
        self.test = test
        self.then = then
        self.else_ = else_
```

FIGURE 2. The Python classes for P_1 AST nodes.

4.4. Compiling polymorphism

As discussed earlier, a Python expression may result in different types of values and that the type may be determined during program execution (at run-time). In general, the ability of a language to allow multiple types of values to be returned from the same expression, or be stored at the same location in memory, is called *polymorphism*. The following is the dictionary definition for this word.

> **pol•y•mor•phism**
> noun
> the occurrence of something in several different forms

The term "polymorhism" can be remembered from its Greek roots: "poly" means "many" and "morph" means "form".

Recall the following example of polymorphism in Python.

```
2 if input() else [1, 2, 3]
```

This expression sometimes results in the integer `2` and sometimes in the list `[1, 2, 3]`.

```
>>> 2 if input() else [1, 2, 3]
1
2
>>> 2 if input() else [1, 2, 3]
0
[1, 2, 3]
```

Consider how the following program would be flattened into a sequence of statements by our compiler.

```
print 2 if input() else [1, 2, 3]
```

We introduce a temporary variable `tmp1` which could point to either an integer or a list depending on the input.

```
tmp0 = input()
if tmp0:
  tmp1 = 2
else:
  tmp1 = [1, 2, 3]
print tmp1
```

Thinking further along in the compilation process, we end up assigning variables to registers, so we'll need a way for a register to refer to either an integer or a list. Note that in the above, when we print `tmp1`, we'll need some way of deciding whether `tmp1` refers to an integer or a list. Also, note that a list could require many more bytes than what could fit in a registers.

One common way to deal with polymorphism is called *boxing*. This approach places all values on the heap and passes around pointers to values in registers. A pointer has the same size regardless of what it points to, and a pointer fits into a register, so this provides a simple solution to the polymorphism problem. When allocating a value on the heap, some space at the beginning is reserved for a tag (an integer) that says what type of value is stored there. For example, the tag `0` could mean that the following value is an integer, `1` means that the value is a Boolean, etc.

Boxing comes with a heavy price: it requires accessing memory which is extremely slow on modern CPUs relative to accessing values from registers. Suppose a program just needs to add a couple integers. Written directly in x86 assembly, the two integers would be stored in registers and the addition instruction would work directly

on those registers. In contrast, with boxing, the integers must be first loaded from memory, which could take 100 or more cycles. Furthermore, the space needed to store an integer has doubled: we store a pointer and the integer itself.

To speed up common cases such as integers and arithmetic, we can modify the boxing approach as follows. Instead of allocating integers on the heap, we can instead go ahead and store them directly in a register, but reserve a couple bits for a tag that says whether the register contains an integer or whether it contains a pointer to a larger value such as a list. This technique is somewhat questionable from a correctness perspective as it reduces the range of plain integers that we can handle, but it provides such a large performance improvement that it is hard to resist.

We will refer to the particular polymorphic representation suggested in these notes as `pyobj`. The file `runtime.c` includes several functions for working with `pyobj`, and those functions can provide inspiration for how you can write x86 assembly that works with `pyobj`. The two least-significant bits of a `pyobj` are used for the tag; the following C function extracts the tag from a `pyobj`.

```
typedef long int pyobj;
#define MASK 3      /* 3 is 11 in binary */
int tag(pyobj val) { return val & MASK; }
```

The following two functions check whether the `pyobj` contains an integer or a Boolean.

```
#define INT_TAG 0      /* 0 is 00 in binary */
#define BOOL_TAG 1     /* 1 is 01 in binary */
int is_int(pyobj val) { return (val & MASK) == INT_TAG; }
int is_bool(pyobj val) { return (val & MASK) == BOOL_TAG; }
```

If the value is too big to fit in a register, we set both tag bits to `1` (which corresponds to the decimal 3).

```
#define BIG_TAG 3    /* 3 is 11 in binary */
int is_big(pyobj val) { return (val & MASK) == BIG_TAG; }
```

The tag pattern `10` is reserved for later use.

The following C functions in `runtime.c` provide a way to convert from integers and Boolean values into their `pyobj` representation. The idea is to move the value over by 2 bits (losing the top two bits) and then stamping the tag into those 2 bits.

```
#define SHIFT 2
pyobj inject_int(int i) { return (i << SHIFT) | INT_TAG; }
pyobj inject_bool(int b) { return (b << SHIFT) | BOOL_TAG; }
```

The next set of C functions from `runtime.c` provide a way to extract an integer or Boolean from its `pyobj` representation. The idea is simply to shift the values back over by 2, overwriting the tag bits. Note that before applying one of these projection functions, you should first check the tag so that you know which projection function should be used.

```
int project_int(pyobj val) { return val >> SHIFT; }
int project_bool(pyobj val) { return val >> SHIFT; }
```

The following C structures define the heap representation for big values. The `hashtable` structure is defined in the provided hashtable C library.

```
enum big_type_tag { LIST, DICT };

struct list_struct {
  pyobj* data;
  unsigned int len;
};
typedef struct list_struct list;

struct pyobj_struct {
  enum big_type_tag tag;
  union {
    struct hashtable* d;
    list l;
  } u;
};
typedef struct pyobj_struct big_pyobj;
```

When we grow the subset of Python to include more features, such as functions and objects, the alternatives within `big_type_tag` will grow as will the union inside of `pyobj_struct`.

The following C functions from `runtime.c` provide a way to convert from `big_pyobj*` to `pyobj` and back again.

```
pyobj inject_big(big_pyobj* p) { return ((long)p) | BIG_TAG; }
big_pyobj* project_big(pyobj val)
    { return (big_pyobj*)(val & ~MASK); }
```

The `inject_big` function above reveals why we chose to use two bits for tagging. It turns out that on Linux systems, `malloc` always aligns newly allocated memory at addresses that are multiples of four. This means that the two least significant bits are always zero! Thus, we can use that space for the tag without worrying about destroying

the address. We can simply zero-out the tag bits to get back a valid address.

The `runtime.c` file also provides a number of C helper functions for performing arithmetic operations and list/dictionary operations on `pyobj`.

```
int is_true(pyobj v);
void print_any(pyobj p);
pyobj input_int();
big_pyobj* create_list(pyobj length);
big_pyobj* create_dict();
pyobj set_subscript(pyobj c, pyobj key, pyobj val);
pyobj get_subscript(pyobj c, pyobj key);
big_pyobj* add(big_pyobj* x, big_pyobj* y);
int equal(big_pyobj* x, big_pyobj* y);
int not_equal(big_pyobj* x, big_pyobj* y);
```

You will need to generate code to do tag testing, to dispatch to different code depending on the tag, and to inject and project values from `pyobj`. We recommend accomplishing this by adding a new compiler pass after parsing and in front of flattening. For lack of a better name, we call this the 'explicate' pass because it makes explicit the types and operations.

4.5. The explicate pass

As we have seen in Section 4.4, compiling polymorphism requires a representation at run time that allows the code to dispatch between operations on different types. This dispatch is enabled by using *tagged values*.

At this point, it is helpful to take a step back and reflect on why polymorphism in P_1 causes a large shift in what our compilers must do as compared to compiling P_0 (which is completely monomorphic). Consider the following assignment statement:

(4.1)
```
y = x + y
```

As a P_0 program, when our compiler sees the `x + y` expression at *compile time*, it knows immediately that `x` and `y` must correspond to integers at *run time*. Therefore, our compiler can select following x86 instruction to implement the above assignment statement.

```
addl x, y
```

This x86 instruction has the same run-time behavior as the above assignment statement in P_0 (i.e., they are *semantically equivalent*).

Now, consider the example assignment statement (4.1) again but now as a P_1 program. At compile time, our compiler has no way to know whether `x + y` corresponds to an integer addition, a list concatenation, or an ill-typed operation. Instead, it must generate code that makes the decision on which operation to perform at *run time*. In a sense, our compiler can do less at compile now: it has less certain information at compile time and thus must generate code to make decisions at run time. Overall, we are trading off execution speed with flexibility with the introduction of polymorphic operations in our input language.

To provide intuition for this trade off, let us consider a real world analogy. Suppose you a planning a hike for some friends. There are two routes that you are considering (let's say the two routes share the same initial path and fork at some point). Basically, you have two choices: you can decide on a route before the hike (at "plan time") or you can wait to make the decision with your friends during the hike (at "hike time"). If you decide beforehand at plan time, then you can simplify your planning; for example, you can input GPS coordinates for the route on which you decided and leave your map for the other route at home. If you want to be more flexible and decide the route during the hike, then you have to bring maps for both routes in order to have sufficient information to make at hike time. The analogy to your compiler is that to be more flexible at run time ($\sim$ hike time), then your compilation ($\sim$ hike planning) requires you to carry tag information at run time ($\sim$ a map at hike time).

Returning to compiling P_1, Section 4.4 describes how we will represent the run-time tags. The purpose of the explicate pass is to generate the dispatching code (i.e., the decision making code). After the explicate pass is complete, the *explicit AST* that is produced will make explicit operations on integers and Booleans. In other words, all operations that remain will be apply to integers, Booleans, or `big_pyobj*`s. Let us focus on the polymorphic + operation in the the example assignment statement (4.1). The AST produced by the parser is as follows:

(4.2)
```
Add((Name('x'), Name('y'))) .
```

We need to create an AST that captures deciding which "+" operation to use based on the run-time types of `x` and `y`.

For +, we have three possibilities: integer addition, list concatenation, or type error. We want a case for integer addition, as we can implement that operation efficiently with an `addl` instruction. To decide whether we have list concatenation or error, we decide to leave

that dispatch to a call in `runtime.c`, as a list concatenation is expensive anyway (i.e., requires going to memory). The `add` function

```
big_pyobj* add(big_pyobj* x, big_pyobj* y)
```

in `runtime.c` does exactly what is described here. To represent the two cases in an explicit AST, we will reuse the `Add` node for *integer* addition and a `CallFunc` node to `add` for the `big_pyobj*` addition. Take note that the `Add` node before the explicate pass represents the polymorphic + of P_1, but it represents integer addition in an explicit AST after the explicate pass. Another choice could have been to create an `IntegerAdd` node kind to make it clear that it applies only to integers.

Now that we have decided which node kinds will represent which + operations, we know that the explicit AST for expression (4.2) is informally as follows:

```
IfExp(
  tag of Name('x') is 'int or bool'
    and tag of Name('y') is 'int or bool',

  convert back to 'pyobj'
    Add(convert to 'int' Name('x'), convert to 'int' Name('y')),

  IfExp(
    tag of Name('x') is 'big'
      and tag of Name('y') is 'big',

    convert back to 'pyobj'
      CallFunc(Name('add'),
        [convert to 'big' Name('x'), convert to 'big' Name('y')]),

    CallFunc(... abort because of run-time type error ...)

  )

)
```

Looking at the above explicit AST, our generated code will at run time look at the tag of the polymorphic values for `x` and `y` to decide whether it is an integer add (i.e., `Add(...)`) or a `big_pyobj*` add (i.e., `CallFunc(Name('add'), ...)`).

What are these "convert" operations? Recall that at run time we need a polymorphic representation of values (i.e., some 32-bit value that can be an 'int', 'bool', 'list', or 'dict'), which we call `pyobj`. It is a `pyobj` that has a tag. However, the integer add at run time (which

corresponds to the `Add` AST node here at compile time) should take "pure" integer arguments (i.e., without tags). Similar, the `add` function call takes `big_pyobj*` arguments (not `pyobj`). We need to generate code that converts `pyobj`s to other types at appropriate places. From Section 4.4, we have described how we get the integer, boolean, or `big_pyobj*` part from a `pyobj` by shifting or masking. Thus, for "convert to whatever" in the above, we need insert AST nodes that represent these *type conversions*. To represent these new type conversion operations, we recommend creating two new AST classes: `ProjectTo` that represents converting from `pyobj` *to* some other type and `InjectFrom` that represents converting to `pyobj` *from* some other type. Analogously, we create a `GetTag` AST class to represent tag lookup, which will be implemented with the appropriate masking.

Note that we might have ben tempted to insert AST nodes that represent directly shifting or masking. While this choice could work, we choose to separate the conceptual operation (i.e., type conversion) from the actual implementation mechanism (i.e., shifting or masking). Our instruction selection phase will implement `ProjectTo` and `InjectFrom` with the appropriate shift or masking instructions.

As a compiler writer, there is one more concern in implementing the explicate pass. Suppose we are implementing the case for explicating `Add`, that is, we are implementing the transformation in general for

`Add((`e_1`, `e_2`))`

where e_1, e_2 are arbitrary subexpressions. Observe in the explicit AST example above, `Name('x')` corresponds to e_1 and `Name('y')` to e_2. Furthermore, observe that `Name('x')` and `Name('y')` each appear four times in the output explicit AST. If instead of `Name`s, we have arbitrary expressions and duplicate them in the same manner, we run into correctness issues. In particular, if e_1 or e_2 are side-effecting expressions (i.e., include `input()`), then duplicating them would change the semantics of the program (e.g., we go from reading once to multiple times). Thus, we need to evaluate the subexpressions once before duplicating them, that is, we can bind the subexpressions to `Name`s and then use the `Name`s in their place.

We introduce a `Let` construct for this purpose:

```
Let(var, rhs, body) .
```

The `Let` construct is needed so that you can use the result of an expression multiple times without duplicating the expression itself, which would duplicate its effects. The semantics of the `Let` is that

```
class GetTag(Node):
    def __init__(self, arg):
        self.arg = arg

class InjectFrom(Node):
    def __init__(self, typ, arg):
        self.typ = typ
        self.arg = arg

class ProjectTo(Node):
    def __init__(self, typ, arg):
        self.typ = typ
        self.arg = arg

class Let(Node):
    def __init__(self, var, rhs, body):
        self.var = var
        self.rhs = rhs
        self.body = body
```

FIGURE 3. New internal AST classes for the output of the explicate pass.

the `rhs` should be evaluated and then assigned to the variable `var`. Then the `body` should be evaluated where the `body` can refer to the variable. For example, the expression

```
Add(( Add((Const(1), Const(2))) , Const(3) ))
```

should evaluate to the same value as

```
Let( Name('x'), Add((Const(1), Const(2))),
     Add(( Name('x') , Const(3) )) )
```

(i.e., they are equivalent semantically).

Overall, to represent the new operations in your abstract syntax trees, we recommend creating the new AST classes in Figure 3.

Exercise 4.3. Implement an `explicate` pass that takes a P_1 AST with polymorphic operations and explicates it to produce an explicit AST where all such polymorphic operations have been transformed to dispatch code to monomorphic operations.

4.6. Type checking the explicit AST

A good way to catch errors in your compiler is to check whether the type of value produced by every expression makes sense. For

example, it would be an error to have a projection nested inside of another projection:

```
ProjectTo('int',
  ProjectTo('int', InjectFrom('int', Const(1)))
)
```

The reason is that projection expects the subexpression to be a `pyobj`.

What we describe in this section is a type checking phase applied to the explicit AST produced as a sanity check for your explicate pass. The explicate pass can be tricky to get right, so we want to have way to detect errors in the explicate pass before going through the rest of the compiler. Note that the type checking that we describe here does not reject input programs at compile time as we may be used to from using statically-typed languages (e.g., Java). Rather, any type errors that result from using the checker that we describe here points to a bug in the explicate pass.

It is common practice to specify what types are expected by writing down an "if-then" rule for each kind of AST node. For example, the rule for `ProjectTo` is:

> For any expression e and any type T selected from the set { `int`, `bool`, `big` }, if e has type `pyobj`, then `ProjectTo(`T`,` e`)` has type T.

It is also common practice to write "if-then" rules using a horizontal line, with the "if" part written above the line and the "then" part written below the line.

$$\frac{e \text{ has type } \texttt{pyobj} \qquad T \in \{\texttt{int}, \texttt{bool}, \texttt{big}\}}{\texttt{ProjectTo}(T, e) \text{ has type } T}$$

Because the phrase "has type" is repeated so often in these type checking rules, it is abbreviated to just a colon. So the above rule is abbreviated to the following.

$$\frac{e : \texttt{pyobj} \qquad T \in \{\texttt{int}, \texttt{bool}, \texttt{big}\}}{\texttt{ProjectTo}(T, e) : T}$$

The `Let(var, rhs, body)` construct poses an interesting challenge. The variable `var` is assigned the `rhs` and is then used inside `body`. When we get to an occurrence of `var` inside `body`, how do we know what type the variable will be? The answer is that we need a dictionary to map from variable names to types. A dictionary used for this purpose is usually called an *environment* (or in older books, a *symbol table*). The capital Greek letter gamma, written Γ, is typically used for referring to environments. The notation $\Gamma, x : T$ stands for

making a copy of the environment Γ and then associating T with the variable x in the new environment. The type checking rules for `Let` and `Name` are therefore as follows.

$$\frac{e_1 : T_1 \text{ in } \Gamma \qquad e_2 : T_2 \text{ in } \Gamma, x : T_1}{\texttt{Let}(x, e_1, e_2) : T_2 \text{ in } \Gamma} \qquad \frac{\Gamma[x] = T}{\texttt{Name}(x) : T \text{ in } \Gamma}$$

Type checking has roots in logic, and logicians have a tradition of writing the environment on the left-hand side and separating it from the expression with a turn-stile ($\vdash$). The turn-stile does not have any intrinsic meaning per se. It is punctuation that separates the environment Γ from the expression e. So the above typing rules are commonly written as follows.

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma, x : T_1 \vdash e_2 : T_2}{\Gamma \vdash \texttt{Let}(x, e_1, e_2) : T_2} \qquad \frac{\Gamma[x] = T}{\Gamma \vdash \texttt{Name}(x) : T}$$

Overall, the statement $\Gamma \vdash e : T$ is an example of what is called a *judgment*. In particular, this judgment says, "In environment Γ, expression e has type T." Figure 4 shows the type checking rules for all of the AST classes in the explicit AST.

Exercise 4.4. Implement a type checking function that makes sure that the output of the explicate pass follows the rules in Figure 4. Also, extend the rules to include checks for statements.

4.7. Update expression flattening

The output AST from the explicate pass contains a number of new AST classes that were not handled by the `flatten` function from chapter 1. The new AST classes are `IfExp`, `Compare`, `Subscript`, `GetTag`, `InjectFrom`, `ProjectTo`, and `Let`. The `Let` expression simply introduces an extra assignment, and therefore no `Let` expressions are needed in the output. When flattening the `IfExp` expression, I recommend using an `If` statement to represent the control flow in the output. Alternatively, you could reduce immediately to labels and jumps, but that makes liveness analysis more difficult. In liveness analysis, one needs to know what statements can preceed a given statement. However, in the presense of jump instructions, you would need to build an explicit control flow graph in order to know the preceeding statements. Instead, I recommend postponing the reduction to labels and jumps to after register allocation, as discussed below in Section 4.10.

Exercise 4.5. Update your `flatten` function to handle the new AST classes. Alternatively, rewrite the `flatten` function into a visitor

$$\frac{n \text{ is an integer}}{\Gamma \vdash \texttt{Const}(n) : \texttt{int}} \qquad \frac{b \text{ is a Boolean}}{\Gamma \vdash \texttt{Const}(b) : \texttt{bool}} \qquad \frac{\Gamma[x] = T}{\Gamma \vdash \texttt{Name}(x) : T}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash \texttt{Add}(e_1, e_2) : \texttt{int}} \qquad \frac{\Gamma \vdash e : \texttt{int}}{\Gamma \vdash \texttt{UnarySub}(e) : \texttt{int}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \quad \Gamma \vdash e_2 : T \quad \Gamma \vdash e_3 : T}{\Gamma \vdash \texttt{IfExp}(e_1, e_2, e_3) : T}$$

$$\frac{\Gamma \vdash e : T \quad T \in \{\texttt{int}, \texttt{bool}, \texttt{big}\}}{\Gamma \vdash \texttt{InjectFrom}(T, e) : \texttt{pyobj}} \qquad \frac{\Gamma \vdash e : \texttt{pyobj} \quad T \in \{\texttt{int}, \texttt{bool}, \texttt{big}\}}{\Gamma \vdash \texttt{ProjectTo}(T, e) : T}$$

$$\frac{\Gamma \vdash e : \texttt{pyobj}}{\Gamma \vdash \texttt{GetTag}(e) : \texttt{int}} \qquad \frac{\Gamma \vdash e_1 : \texttt{pyobj} \quad \Gamma \vdash e_2 : \texttt{pyobj}}{\Gamma \vdash \texttt{Compare}(e_1, [(\texttt{is}, e_2)]) : \texttt{bool}}$$

$$\frac{\Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T \quad T \in \{\texttt{int}, \texttt{bool}\} \quad op \in \{\texttt{==}, \texttt{!=}\}}{\Gamma \vdash \texttt{Compare}(e_1, [(op, e_2)]) : \texttt{bool}}$$

$$\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma, x : T_1 \vdash e_2 : T_2}{\Gamma \vdash \texttt{Let}(x, e_1, e_2) : T_2}$$

$$\frac{\Gamma \vdash e_1 : \texttt{pyobj} \quad \Gamma \vdash e_2 : \texttt{pyobj}}{\Gamma \vdash \texttt{Subscript}(e_1, e_2) : \texttt{pyobj}}$$

FIGURE 4. Type checking rules for expressions in the explicit AST.

class and then create a new visitor class that inherits from it and that implements visit methods for the new AST nodes.

4.8. Update instruction selection

The instruction selection phase should be updated to handle the new AST classes `If`, `Compare`, `Subscript`, `GetTag`, `InjectFrom`, and `ProjectTo`. Consult Appendix 6.4 for suggestions regarding which x86 instructions to use for translating the new AST classes. Also, you will need to update the function call for printing because you should now use the `print_any` function.

Exercise 4.6. Update your instruction selection pass to handle the new AST classes.

4.9. Update register allocation

Looking back at Figure 5, there are several sub-passes within the register allocation pass, and each sub-pass needs to be updated to deal with the new AST classes.

In the liveness analysis, the most interesting of the new AST classes is the `If` statement. What liveness information should be propagated into the "then" and "else" branch and how should the results from the two branches be combined to give the result for the entire `If`? If we could somehow predict the result of the test expression, then we could select the liveness results from one branch or the other as the results for the `If`. However, its impossible to predict this in general (e.g., the test expression could be `input()`)), so we need to make a conservative approximation: we assume that either branch could be taken, and therefore we consider a variable to be live if it is live in either branch.

The code for building the interference graph needs to be updated to handle `If` statements, as does the code for finding all of the local variables. In addition, you need to account for the fact that the register `al` is really part of register `eax` and that register `cl` is really part of register `ecx`.

The graph coloring algorithm itself works on the interference graph and not the AST, so it does not need to be changed.

The spill code generation pass needs to be updated to handle `If` statements and the new x86 instructions that you used in the instruction selection pass.

Similarly, the code for assigning homes (registers and stack locations) to variables must be updated to handle `If` statements and the new x86 instructions.

4.10. Removing structured control flow

Now that register allocation is finished, and we no longer need to perform liveness analysis, we can lower the `If` statements down to x86 assembly code by replacing them with a combination of labels and jumps. The following is a sketch of the transformation from `If` AST nodes to labels and jumps.

`if` x`:`
  then instructions
`else:`
  else instructions

$\Longrightarrow$

```
cmpl $0, x
je else_label_5
then instructions
jmp end_label_5
else_label_5:
else instructions
end_label_5:
```

Exercise 4.7. Write a compiler pass that removes `If` AST nodes, replacing them with combinations of labels and jumps.

4.11. Updates to print x86

You will need to update the compiler phase that translates the x86 intermediate representation into a string containing the x86 assembly code, handling all of the new instructions introduced in the instruction selection pass and the above pass that removes `If` statements.

Putting all of the above passes together, you should have a complete compiler for P_1.

Exercise 4.8. Extend your compiler to handle the P_1 subset of Python. You may use the parser from Python's `compiler` module, or for extra credit you can extend your own parser. Figure 5 shows the suggested organization for your compiler.

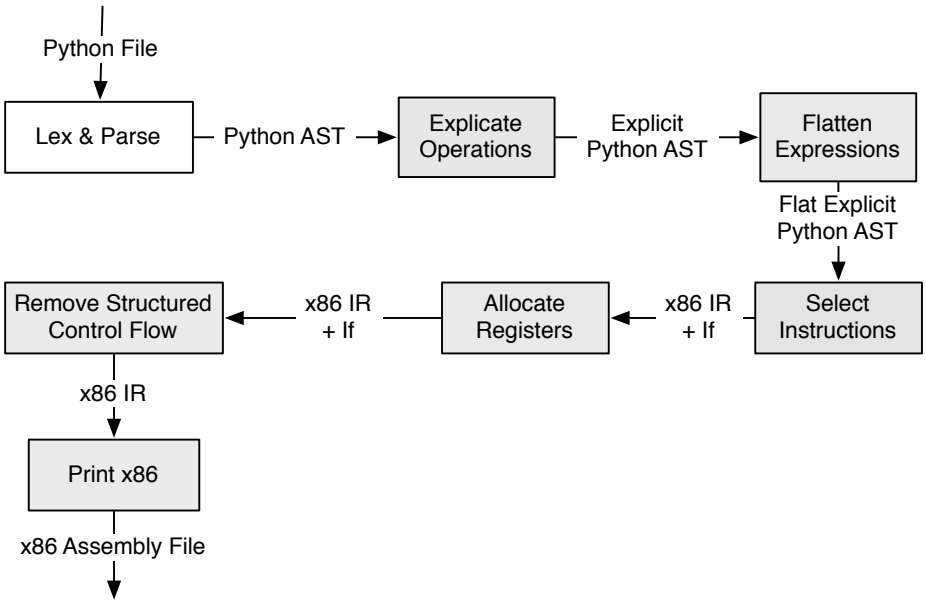


FIGURE 5. Overview of the compiler organization.

CHAPTER 5

Functions

The main ideas in this chapter are:

first-class functions: functions are values that can be passed as arguments to other functions, returned from functions, stored in lists and dictionaries, assigned to variables, etc.

lexical scoping of variables: scopes separate variables with the same name; lexical scoping dictates that a variable reference is resolved by looking at its lexical environment.

5.1. Syntax of P_2

We introduce two constructs for creating functions: the `def` statement and the `lambda` expression. We also add an expression for calling a function with some arguments. To keep things manageable, we leave out function calls with keyword arguments. The concrete syntax of the P_2 subset of Python is shown in Figure 1. Figure 2 shows the additional Python classes for the P_2 AST.

```
expression ::= expression "(" expr_list ")"
             | "lambda" id_list ":" expression
id_list ::= ϵ | identifier | identifier "," id_list
simple_statement ::= "return" expression
statement ::= simple_statement
            | compound_stmt
compound_stmt ::= "def" identifier "(" id_list ")" ":" suite
suite ::= "\n" INDENT statement+ DEDENT
module ::= statement+
```

FIGURE 1. Concrete syntax for the P_2 subset of Python. (In addition to that of P_1.)

5.2. Semantics of P_2

Functions provide an important mechanism for reusing chunks of code. If there are several places in a program that compute the same thing, then the common code can be placed in a function and

```
class CallFunc(Node):
    def __init__(self, node, args):
        self.node = node
        self.args = args

class Function(Node):
    def __init__(self, decorators, name, argnames, defaults, \
                 flags, doc, code):
        self.decorators = decorators # ignore
        self.name = name
        self.argnames = argnames
        self.defaults = defaults     # ignore
        self.flags = flags           # ignore
        self.doc = doc               # ignore
        self.code = code

class Lambda(Node):
    def __init__(self, argnames, defaults, flags, code):
        self.argnames = argnames
        self.defaults = defaults     # ignore
        self.flags = flags           # ignore
        self.code = code

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

FIGURE 2. The Python classes for P_2 ASTs.

then called from many locations. The example below defines and calls a function. The `def` statement creates a function and gives it a name.

```
>>> def sum(l, i, n):
...     return l[i] + sum(l, i + 1, n) if i != n \
...            else 0
...
>>> print sum([1,2,3], 0, 3)
6
>>> print sum([4,5,6], 0, 3)
15
```

Functions are *first class,* which means they are treated just like other values: they may be passed as arguments to other functions, returned from functions, stored within lists, etc.. For example, the `map` function defined below has a parameter `f` that is applied to every element of the list `l`.

```
>>> def map(f, l, i, n):
...     return [f(l[i])] + map(f, l, i + 1, n) if i != n else []
```

Suppose we wish to square every element in an array. We can define a `square` function and then use `map` as follows.

```
>>> def square(x):
...     return x * x
...
>>> print map(square, [1,2,3], 0, 3)
[1, 4, 9]
```

The `lambda` expression creates a function, but does not give it a name. Anonymous functions are handy in situations where you only use the function in one place. For example, the following code uses a `lambda` expression to tell the `map` function to add one to each element of the list.

```
>> print map(lambda x: x + 1, [1,2,3], 0, 3)
[2, 3, 4]
```

Functions may be nested within one another as a consequence of how the grammar is defined in Figure 1. Any statement may appear in the body of a `def` and any expression may appear in the body of a `lambda`, and functions may be created with statements or expressions. Figure 3 shows an example where one function is defined inside another function.

```
>>> def f(x):
...     y = 4
...     return lambda z: x + y + z
...
>>> f1 = f(1)
>>> print f1(3)
8
```

FIGURE 3. An example of a function nested inside another function.

A function may refer to parameters and variables in the surrounding scopes. In the above example, the `lambda` refers to the `x` parameter and the `y` local variable of the enclosing function `f`.

One of the trickier aspects of functions in Python is their interaction with variables. A function definition introduces a new scope. A variable *assignment* within a function also declares that variable within the function's scope. So, for example, in the following code, the scope of the variable `a` is the body of function `f` and not the global scope.

```
>>> def f():
...   a = 2
...   return a
>>> f()
2
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'a' is not defined
```

Python's rules about variables can be somewhat confusing when a variable is assigned in a function and has the same name as a variable that is assigned outside of the function. For example, in the following code the assignment `a = 2` does not affect the variable `a` in the global scope but instead introduces a new variable within the function `g`.

```
>>> a = 3
>>> def g():
...     a = 2
...     return a
>>> g()
2
>>> a
3
```

An assignment to a variable anywhere within the function body introduces the variable into the scope of the entire body. So, for example, a reference to a variable before it is assigned will cause an error (even if there is a variable with the same name in an outer scope).

```
>>> a = 3
>>> def h():
...     b = a + 2
...     a = 1
...     return b + a
>>> h()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 2, in h
UnboundLocalError: local variable 'a' referenced before
  assignment
```

Exercise 5.1. Write five programs in the P_2 subset of Python that help you understand the language. Look for corner cases or unusual aspects of the language to test in your programs.

5.3. Overview of closure conversion

The major challenge in compiling Python functions to x86 is that functions may not be nested in x86 assembly. Therefore we must unravel the nesting and define each function at the top level. Moving the function definitions is straightforward, but it takes a bit more work to make sure that the function behaves the same way it use to, after all, many of the variables that were in scope at the point where the function was originally defined are not in scope at the top level.

When we move a function, we have to worry about the variables that it refers to that are not parameters or local variables. We say that a variable reference is *bound* with respect to a given expression or statement, let's call it P, if there is an function or lambda inside P that encloses the variable reference and that function or lambda has that variable as a parameter or local. We say that a variable is *free* with respect to an expression or statement P if there is a reference to the variable inside P that is not bound in P. In the following, the variables `y` and `z` are bound in function `f`, but the variable `x` is free in function `f`.

```
x = 3
def f(y):
  z = 3
  return x + y + z
```

The definition of free variables applies in the same way to nested functions. In the following, the variables `x`, `y`, and `z` are free in the `lambda` expression whereas `w` is bound in the `lambda` expression. The variable `x` is free in function `f`, whereas the variables `w`, `y`, and `z` are bound in `f`.

```
x = 3
def f(y):
  z = 3
  return lambda w: x + y + z + w
```

Figure 4 gives part of the definition of a function that computes the free variables of an expression. Finishing this function and defining a similar function for statements is left to you.

The process of *closure conversion* turns a function with free variables into an behaviorally equivalent function without any free variables. A function without any free variables is called "closed", hence the term "closure conversion". The main trick in closure conversion is to turn each function into a value that contains a pointer to the function and a list that stores the values of the free variables. This

```
def free_vars(n):
    if isinstance(n, Const):
        return set([])
    elif isinstance(n, Name):
        if n.name == 'True' or n.name == 'False':
            return set([])
        else:
            return set([n.name])
    elif isinstance(n, Add):
        return free_vars(n.left) | free_vars(n.right)
    elif isinstance(n, CallFunc):
        fv_args = [free_vars(e) for e in n.args]
        free_in_args = reduce(lambda a, b: a | b, fv_args, set([]))
        return free_vars(n.node) | free_in_args
    elif isinstance(n, Lambda):
        return free_vars(n.code) - set(n.argnames)
    ...
```

FIGURE 4. Computing the free variables of an expression.

value is called a *closure* and you'll see that `big_pyobj` has been expanded to include a function inside the union. In the explanation below, we'll use the runtime function *create_closure* to construct a closure and the runtime functions *get_fun_ptr* and *get_free_vars* to access the two parts of a closure. When a closure is invoked, the free variables list must be passed as an extra argument to the function so that it can obtain the values for free variables from the list.

Figure 5 shows the result of applying closure conversion to the example in Figure 3. The `lambda` expression has been removed and the associated code placed in the `lambda_0` function. For each of the free variables of the `lambda` (`x` and `y`), we add assignments inside the body of `lambda_0` to initialize those variables by subscripting into the `free_vars_0` list. The lambda expression inside `f` has been replaced by a new kind of primitive operation, creating a closure, that takes two arguments. The first argument is the function name and the second is a list containing the values of the free variables. Now when we call the `f` function, we get back a closure. To invoke a closure, we call the closures' function, passing the closure's free variable array as the first argument. The rest of the arguments are the normal arguments from the call site.

Note that we also created a closure for function `f`, even though `f` was already a top-level function. The reason for this is so that at any call site in the program, we can assume that the thing being applied

is a closure and use the above-described approach for translating the function call.

```
def lambda_0(free_vars_0, z):
  y = free_vars_0[0]
  x = free_vars_0[1]
  return x + y + z

def lambda_1(free_vars_1, x):
  y = 4
  return create_closure(lambda_0, [y, x])

f = create_closure(lambda_1, [])

f1 = get_fun_ptr(f)(get_free_vars(f), 1)
print get_fun_ptr(f1)(get_free_vars(f1), 3)
```

FIGURE 5. Closure conversion applied to the example in Figure 3.

5.4. Overview of heapifying variables

Closure conversion, as described so far, copies the values of the free variables into the closure's array. This works as long as the variables are *not* updated by a later assignment. Consider the following program and the output of the `python` interpreter.

```
def f(y):
    return x + y
x = 2
print f(40)
```

The read from variable `x` should be performed when the function is called, and at that time the value of the variable is `2`. Unfortunately, if we simply copy the value of `x` into the closure for function `f`, then the program would incorrectly access an undefined variable.

We can solve this problem by storing the values of variables on the heap and storing just a pointer to the variable's value in the closure's array. The following shows the result of heapification for the above program. Now the variable `x` refers to a one-element list and each reference to `x` has been replaced by a a subscript operation that accesses the first element of the list (the element at index 0).

```
x = [0]
def f(y):
```

```
    return x[0] + y
  x[0] = 2
  print f(40)
```

Applying closure conversion after heapification gives us the following result, which correctly preserves the behavior of the original program.

```
def lambda_0(free_vars_0, y):
  x = free_vars_0[0]
  return x[0] + y

x = [0]
f = create_closure(lambda_0, [x])
x[0] = 2
print get_fun_ptr(f)(get_free_vars(f), 40)
```

5.4.1. Discussion. For simplicity, we heapify any variable that ends up in a closure. While this step feels heavy weight, it is the most uniform and simplest solution to the problem.

There are certainly opportunities for optimization, but in general, they require additional static analysis. For example,

- If you can tell statically that a local variable of function will not be used after the function has returned, then you can stack allocate it instead of heap allocating. A local variable that cannot be used after the function returns is said to be *non-escaping*, and the analysis to determine whether a variable may escape is called *escape analysis*. In Heapify, we perform a simplistic, very conservative escape analysis that says any variable that ends up in closure may escape. This analysis is quite conservative, as we may create a closure that is only used by called functions but never returned nor stored in the heap.
- If you can tell statically that a variable that ends up in a closure is not modified after that closure is created, then you can instead close on the value of the variable instead of its address (avoiding stack or heap allocation).

5.5. Compiler implementation

Figure 6 shows the suggested organization for the compiler for this chapter. The next few subsections outline the new compiler passes and the changes to the other passes.

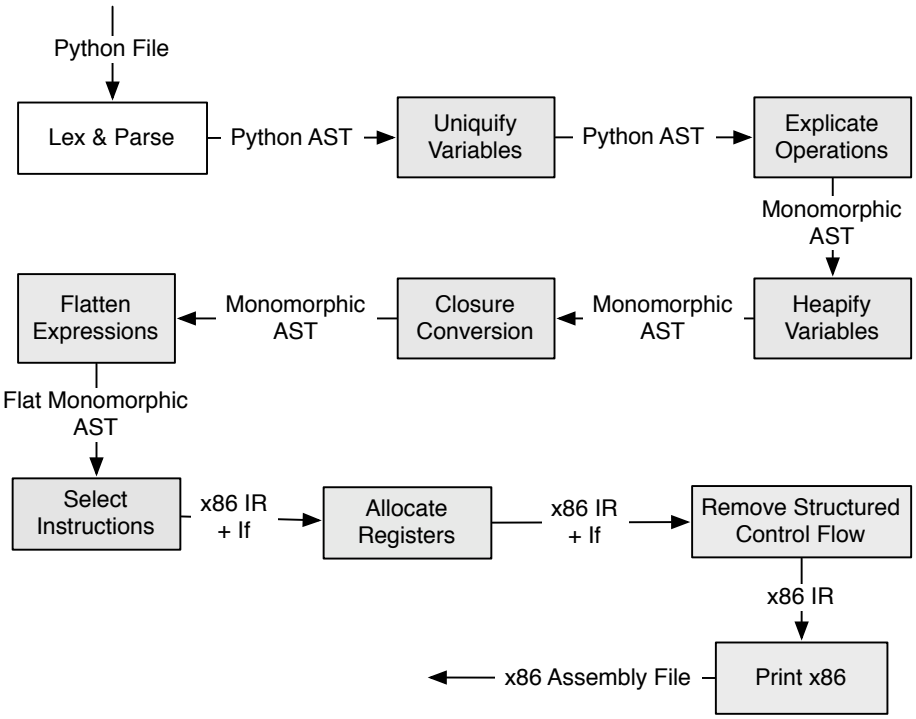


FIGURE 6. Organization of the compiler passes.

5.5.1. The Uniquify Variables Pass. During the upcoming heapify pass, we need to do a fair bit of reasoning about variables, and this reasoning will be a lot easier if we don't have to worry about confusing two different variables because they have the same name. For example, in the following code we do not need to heapify the global `x`, but we do have to heapify the parameter `x`.

```
x = 3
def f(x):
  return lambda y: x + y
print x
```

Thus, the uniquify variables pass renames every variable to make sure that each variable has a unique name.

The main issue to keep in mind when implementing the uniquify variables pass is determining whether a variable use references a variable in the current scope or one in an outer scope. Python's specification is that any variable that is assigned to (statically) is a new local in the current function's scope. Also recall that a `def` $f(\ldots)$`:` $\ldots$

is semantically equivalent to an assignment to a variable f with an anonymous function.

The renaming can be accomplished by incrementing a global counter and use the current value of the counter in the variable name. One strategy is at each new scope, gather all the variables introduced by the scope, compute their renamings (while saving the renamings in a dictionary), and finally apply the renaming to the code.

Uniquifying the above example would result in something like the following:

```
x_0 = 3
def f_1(x_2):
  return lambda y_3: x_2 + y_3
print x_0
```

5.5.2. The Explicate Operations Pass. To simplify the later passes, I recommend converting function definitions and lambda's into a common form. The common form is like a lambda, but has a body that contains a statement instead of an expression. Instead of creating a new AST class, the `Lambda` class can be re-used to represent these new kinds of lambdas. The following is a sketch of the transformation for function definitions.

```
def name(args):
  body
⟹
name = lambda args: body
```

To convert lambdas to the new form, we simply put the body of the lambda in a return statement.

```
lambda args: body
⟹
lambda args: return body
```

5.5.3. The Heapify Variables Pass. To implement this pass we need two helper functions: the function for computing free variables (Figure 4) and a function for determining which variables occur free within nested lambdas. This later function is straightforward to implement. It traverses the entire program, and whenever it encounters a lambda or function definition, it calls the free variables function, removes that functions parameters and local variables from the set of free variables, then marks the remaining variables as needing heapification. I suggest using a dictionary for recording which variables need to be heapified.

Now for the main heapification function. As usual it is a recursive function that traverses the AST. The function returns a new AST. In the following we discuss the most interesting cases.

Lambda: First, compute the local variables of this lambda; I'll name this set L. Let P be the set of parameter names (`argnames`) for this lambda. Make the recursive call on the body of the current lambda. Let *body′* be the result of the recursive call. Then we return a lambda of the following form:

```
lambda P′:
    paramAllocs
    paramInits
    localInits
    body′
```

The list of parameters P' is the same as P except that the parameters that need to be heapified are renamed to new unique names. Let P_h be the parameters in P that need to be heapified. The list of statements *paramAllocs* is a sequence of assignments, each of which assigns a 1-element list to a variable in the set P_h. The list of statements *paramInits* is a sequence of assignments, each of which sets the first element in the list referred to by the variables in P_h to the corresponding renamed parameter in P'. Let L_h be the local variables in L that need to be heapified. The list of statements *localInits* is a sequence of assignments, each of which assigns a 1-element list to a variable in the set L_h.

Name: If the variable x needs to be heapified, then return the expression x`[0]`. Otherwise return x unchanged.

Assign: If the left-hand side of the assignment is a `AssName`, and the variable x needs to be heapified, then return the assignment x`[0] =` rhs', where rhs' has been heapified. Otherwise return the assignment x `=` rhs'.

5.5.4. The Closure Conversion Pass. To implement closure conversion, we'll write a recursive function that takes one parameter, the current AST node, and returns a new version of the current AST node and a list of function definitions that need to be added to the global scope.

Let us look at the two interesting the cases in the closure conversion functions.

Lambda: The process of closure converting lambda expressions is similar to converting function definitions. The lambda expression itself is converted into a closure (an expression creating a two element list) and a function definition for the lambda is returned so that it can be placed in the global scope. So we have

```
lambda params: body
⟹
create_closure(globalname, fvs)
```

where *globalname* is a freshly generated name and *fvs* is a list defined as follows

$$fvs = \text{free\_vars}(body) - params$$

Closure conversion is applied recursively to the *body*, resulting in a *newbody* and a list of function definitions. We return the closure creating expression `create_closure(`*globalname*`,` *fvs*`)` and the list of function definitions, with the following definition appended.

```
def globalname(fvs, params):
  fvs_1 = fvs[0]
  fvs_2 = fvs[1]
  ...
  fvs_n = fvs[n − 1]
  return newbody
```

CallFunc: A function call node includes an expression that should evaluate e_f to a function and the argument expressions $e_1, \ldots, e_n$. Of course, due to closure conversion, e_f should evaluate to a closure object. We therefore need to transform the `CallFunc` so that we obtain the function pointer of the closure and apply it to the free variable list of the closure followed by the normal arguments.

```
e_f(e_1, ..., e_n)
⟹
let tmp = e_f in
  get_fun_ptr(tmp)(get_free_vars(tmp), e_1, ..., e_n)
```

In this pass it is helpful to use a different AST class for indirect function calls, whose operator will be the result of an expression such as above, versus direct calls to the runtime C functions.

5.5.5. The Flatten Expressions Pass. The changes to this pass are straightforward. You just need to add cases to handle functions, return statements, and indirect function calls.

5.5.6. The Select Instructions Pass. You need to update this pass to handle functions, return statements, and indirect function calls. At this point in the compiler it is convenient to create an explicit `main` function to hold all the statements other then the function definitions.

5.5.7. The Register Allocation Pass. The primary change needed in this pass is that you should perform register allocation separately for each function (i.e., you perform liveness analysis, construct an interference graph, and assign registers for each function separately).

Make sure that your Select Instructions pass saves the callee-save registers on the stack in the prologue of each function and restores them in the epilogue. A small optimization would be to wait until after register allocation to decide which callee-save registers need to be saved (rather than always saving all).

5.5.8. The Print x86 Pass. You need to update this pass to handle functions, return statements, and indirect function calls

Exercise 5.2. Extend your compiler to handle P_2.

CHAPTER 6

Objects

The main ideas for this chapter are:

objects and classes: objects are values that bundle together some data (attributes) and some functions (methods). Classes are values that describe how to create objects.

attributes and methods: Both objects and classes can contain attributes and methods. An attribute maps a name to a value and a method maps a name to a function.

inheritance: One class may inherit from one or more other classes, thereby gaining access to the methods in the inherited classes.

6.1. Syntax of P_3

The concrete syntax of P_3 is shown in Figure 1 and the abstract syntax (the Python AST classes) is shown in Figure 2.

```
expression ::= expression "." identifier
expression_list ::= expression ( "," expression )* [","]
statement ::= "class" name ["(" expression_list ")"] ":" suite
            | "if" expression ":" suite "else" ":" suite
            | "while" expression ":" suite
target ::= expression "." identifier
```

FIGURE 1. Concrete syntax for the P_3 subset of Python. (In addition to that of P_2.)

6.2. Semantics of P_3

This week we add a statement for creating classes. For example, the following statement creates a class named `C`.

```
>>> class C:
...     x = 42
```

Assignments in the body of a class create *class attributes*. The above code creates a class `C` with an attribute `x`. Class attributes may be accessed using the dot operator. For example:

```
class AssAttr(Node):
    def __init__(self, expr, attrname, flags):
        self.expr = expr
        self.attrname = attrname
        self.flags = flags          # ignore this

class Class(Node):
    def __init__(self, name, bases, doc, code):
        self.name = name
        self.bases = bases
        self.doc = doc              # ignore this
        self.code = code

class Getattr(Node):
    def __init__(self, expr, attrname):
        self.expr = expr
        self.attrname = attrname

class If(Node):
    def __init__(self, tests, else_):
        self.tests = tests
        self.else_ = else_

class While(Node):
    def __init__(self, test, body, else_):
        self.test = test
        self.body = body
        self.else_ = else_
```

FIGURE 2. The Python classes for P_3 ASTs.

```
>>> print C.x
42
```

The body of a class may include arbitrary statements, including statements that perform I/O. These statements are executed as the class is created.

```
>>> class C:
...     print 4 * 10 + 2
42
```

If a class attribute is a function, then accessing the attribute produces an *unbound method*.

```
>>> class C:
        f = lambda o, dx: o.x + dx
```

```
>>> C.f
<unbound method C.<lambda>>
```

An unbound method is like a function except that the first argument must be an instance of the class from which the method came. We'll talk more about instances and methods later.

Classes are first-class objects, and may be assigned to variables, returned from functions, etc. The following `if` expression evaluates to the class `C`, so the attribute reference evaluates to `42`.

```
>>> class C:
...     x = 42
>>> class D:
...     x = 0

>>> print (C if True else D).x
42
```

6.2.1. Inheritance. A class may inherit from other classes. In the following, class `C` inherits from classes `A` and `B`. When you reference an attribute in a derived class, if the attribute is not in the derived class, then the base classes are searched in depth-first, left-to-right order. In the following, `C.x` resolves to `A.x` (and not `B.x`) whereas `C.y` resolves to `B.y`.

```
>>> class A:
...     x = 4

>>> class B:
...     x = 0
...     y = 2

>>> class C(A, B):
...     z = 3

>>> print C.x * 10 + C.y
42
```

6.2.2. Objects. An object (or *instance*) is created by calling a class as if it were a function.

```
o = C()
```

If the class has an attribute named `__init__`, then once the object is allocated, the `__init__` function is called with the object as it's first argument. If there were arguments in the call to the class, then these arguments are also passed to the `__init__` function.

```
>>> class C:
...     def __init__(o, n):
...         print n

>>> o = C(42)
42
```

An instance may have associated *data attributes*, which are created by assigning to the attribute. Data attributes are accessed with the dot operator.

```
>>> o.x = 7
>>> print o.x
7
```

Different objects may have different values for the same attribute.

```
>>> p = C(42)
42
>>> p.x = 10
>>> print o.x, p.x
7, 10
```

Objects live on the heap and may be aliased (like lists and dictionaries).

```
>>> print o is p
False
>>> q = o
>>> print q is o
True
>>> q.x = 1
>>> print o.x
1
```

A data attribute may be a function (because functions are first class). Such a data attribute is not a method (the object is not passed as the first parameter).

```
>>> o.f = lambda n: n * n
>>> o.f(3)
9
```

When the dot operator is applied to an object but the specified attribute is not present in the object itself, the class of the object is searched followed by the base classes in depth-first, left-to-right order.

```
>>> class C:
...     y = 3
```

```
>>> o = C()
>>> print o.y
3
```

If an attribute reference resolves to a function in the class or base class of an object, then the result is a *bound method*.

```
>>> class C:
...     def move(o,dx):
...         o.x = o.x + dx
>>> o = C()
>>> o.move
<bound method C.move of <__main__.C instance at 0x11d3fd0>>
```

A bound method ties together the receiver object (`o` in the above example) with the function from the class (`move`). A bound method can be called like a function, where the receiver object is implicitly the first argument and the arguments provided at the call are the rest of the arguments.

```
>>> o.x = 40
>>> o.move(2)
>>> print o.x
42
```

Just like everything else in Python, bound methods are first class and may be stored in lists, passed as arguments to functions, etc.

```
>>> mlist = [o.move,o.move,o.move]
>>> i = 0
>>> while i != 3:
...     mlist[i](1)
...     i = i + 1
>>> print o.x
45
```

You might wonder how the Python implementation knows whether to make a normal function call or whether to perform a method call (which requires passing the receiver object as the first argument). The answer is that the implementation checks the type tag in the operator to see whether it is a function or bound method and then treats the two differently.

Exercise 6.1. Read:

(1) Section 9 of the Python Tutorial
(2) Python Language Reference, Section 3.2

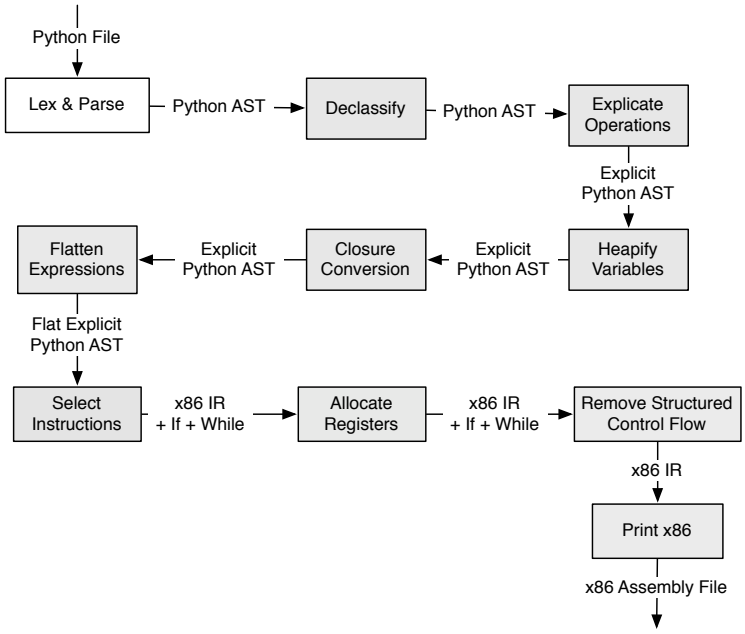


FIGURE 3. Structure of the compiler.

6.2.3. If and While Statements. This chapter we also add `if` and `while` statements. For the `if` statement, you don't need to support `elif` and you can assume that every `if` has an `else`. For `while` statements, you don't need to support the `else` clause.

One of the more interesting aspects of extending your compiler to handle `While` statements is that you'll need to figure out how to propagate the live-variable information through `While` statements in the register allocation phase.

6.3. Compiling Classes and Objects

Figure 3 shows the structure of the compiler with the addition of classes and objects. We insert a new pass at the beginning of the compiler that lowers classes and objects to more primitive operations and then we update the rest of the compiler to handle these new primitives.

In addition to the new passes and primitives, the entities introduced this week are all first-class, so the `big_pyobj` union in `runtime.h` has been extended.

class: The runtime representation for a class stores a list of base classes and a dictionary of attributes.

object: The runtime representation for an object stores its class and a dictionary of attributes.

unbound method: The runtime representation of an unbound method contains the underlying function and the class object on which the attribute access was applied that created the unbound method.

bound method: The runtime representation for a bound method includes the function and the receiver object.

The following are the new functions in `runtime.h` for working with classes, objects, bound methods, and unbound methods.

```
 /* bases should be a list of classes */
big_pyobj* create_class(pyobj bases);
big_pyobj* create_object(pyobj cl);
 /* inherits returns true if class c1 inherits from class c2 */
int inherits(pyobj c1, pyobj c2);
 /* get_class returns the class from an object or unbound method */
big_pyobj* get_class(pyobj o);
 /* get_receiver returns the receiver from inside a bound method */
big_pyobj* get_receiver(pyobj o);
 /* get_function returns the function from inside a method */
big_pyobj* get_function(pyobj o);
int has_attr(pyobj o, char* attr);
pyobj get_attr(pyobj c, char* attr);
pyobj set_attr(pyobj obj, char* attr, pyobj val);
```

6.3.1. Compiling empty class definitions and class attributes. Compiling full class definitions is somewhat involved, so I first recommend compiling empty class definitions. We begin with class definitions that have a trivial body.

```
>>> class C:
...     0
```

The class definition should be compiled into an assignment to a variable named `C`. The right-hand-side of the assignment should be an expression that allocates a class object with an empty hashtable for attributes and an empty list of base classes. So, in general, the transformation should be

```
class C:
  0
⟹
C = create_class()
```

where `create_class` is a new C function in `runtime.h`.

While a class with no attributes is useless in C++, in Python you can add attributes to the class after the fact. For example, we can proceed to write

```
>>> C.x = 3
>>> print C.x
3
```

An assignment such as `C.x = 3` (the `AssAttr` node) should be transformed into a call to `set_attr`. In this example, we would have `set_attr(C, "x", 3)`. Note that this requires adding support for string constants to your intermediate language.

The attribute access `C.x` (the `Getattr` node) in the `print` statement should be translated into a call to the `get_attr` function in `runtime.h`. In this case, we would have `get_attr(C, "x")`.

6.3.2. Compiling class definitions. A class body may contain an arbitrary sequence of statements, and some of those statements (assignments and function definitions) add attributes to the class object. Consider the following example.

```
class C:
    x = 3
    if True:
        def foo(self, y):
            w = 3
            return y + w
        z = x + 9
    else:
        def foo(self, y):
            return self.x + y
    print 'hello world!\n'
```

This class definition creates a class object with three attributes: `x`, `foo`, and `z`, and prints out `hello world!`.

The main trick to compiling the body of a class is to replace assignments and function definitions so that they refer to attributes in the class. The replacement needs to go inside compound statements such as `If` and `While`, but not inside function bodies, as those assignments correspond to local variables of the function. One can imagine transforming the above code to something like the following:

```
class C:
    pass
C.x = 3
if True:
    def __foo(self, y):
```

```
        w = 3
        return y + w
    C.foo = __foo
    C.z = C.x + 9
else:
    def __foo(self, y):
        return self.x + y
    C.foo = __foo
print 'hello world!\n'
```

Once the code is transformed as above, the rest of the compilation passes can be applied to it as usual.

In general, the translation for class definitions is as follows.

`class` C`(`$B_1, \ldots, B_n$`):`
  body
$\Longrightarrow$
tmp `= create_class([`$B_1, \ldots, B_n$`])`
newbody
C `=` *tmp*

Instead of assigning the class to variable C, we instead assign it to a unique temporary variable and then assign it to C after the *newbody*. The reason for this is that the scope of the class name C does not include the body of the class.

The *body* is translated to *newbody* by recursively applying the following transformations. You will need to know which variables are assigned to (which variables are class attributes), so before transforming the *body*, first find all the variables assigned-to in the *body* (but not assigned to inside functions in the *body*).

The translation for assignments is:

x `=` e
$\Longrightarrow$
`set_attr(`*tmp*`, "`x`", `e'`)`

where e' is the recursively processed version of e.

The translation for variables is somewhat subtle. If the variable is one of the variables assigned somewhere in the body of this class, and if the variable is also in scope immediately outside the class, then translate the variable into a conditional expression that either does an attribute access or a variable access depending on whether the attribute is actually present in the class value.

x
$\Longrightarrow$
`get_attr(`*tmp*`, "`x`") if has_attr(`*tmp*`, "`x`") else` x

If the variable is assigned in the body of this class but is not in scope outside the class, then just translate the variable to an attribute access.

x
$\Longrightarrow$
`get_attr(`tmp`, "`x`")`

If the variable is not assigned in the body of this class, then leave it as a variable.

x
$\Longrightarrow$
x

The translation for function definitions is:

`def` f`(`e_1`,...,`e_n`):`
    $body$
$\Longrightarrow$
`def` f_tmp`(`e_1`,...,`e_n`):`
    $body$ # the body is unchanged, class attributes are not in scope here
`set_attr(`tmp`, "`f`", `f_tmp`)`

6.3.3. Compiling objects. The first step in compiling objects is to implement object construction, which in Python is provided by invoking a class as if it were a function. For example, the following creates an instance of the `C` class.

```
C()
```

In the AST, this is just represented as a function call (`CallFunc`) node. Furthermore, in general, at the call site you won't know at compile-time that the operator is a class object. For example, the following program might create an instance of class `C` or it might call the function `foo`.

```
def foo():
    print 'hello world\n'

(C if input() else foo)()
```

This can be handled with a small change to how you compile function calls. You will need to add a conditional expression that checks whether the operator is a class object or a function. If it is a class object, you need to allocate an instance of the class. If the class defines an `__init__` method, the method should be called immediately after the object is allocated. If the operator is not a class, then perform a function call.

In the following we describe the translation of function calls. The Python `IfExp` is normally written as e_1 `if` e_0 `else` e_2 where e_0 is the condition, e_1 is evaluated if e_0 is true, and e_2 is evaluated if e_0 is false. I'll instead use the following textual representation:

```
if e0 then e1 else e2
```

In general, function calls can now be compiled like this:

```
e0(e1,...,en)
⟹
let f = e0 in
let a1 = e1 in
  ⋮
let an = en in
if is_class(f) then
    let o = create_object(f) in
    if has_attr(f, '__init__') then
        let ini = get_function(get_attr(f, '__init__')) in
        let _ = ini(o, a1,...,an) in
          o
    else o
else
    f(a1,...,an)          # normal function call
```

The next step is to add support for creating and accessing attributes of an object. Consider the following example.

```
o = C()
o.w = 42

print o.w
print o.x     # attribute from the class C
```

An assignment to an attribute should be translated to a call to `set_attr` and accessing an attribute should be translated to a call to `get_attr`.

6.3.4. Compiling bound and unbound method calls. A call to a bound or unbound method also shows up as a function call node (`CallFunc`) in the AST, so we now have four things that can happen at a function call (we already had object construction and normal function calls). To handle bound and unbound methods, we just need to add more conditions to check whether the operator is a bound or unbound method. In the case of an unbound method, you should call the underlying function from inside the method. In the case of a bound method, you call the underlying function, passing

the receiver object (obtained from inside the bound method) as the first argument followed by the normal arguments. The suggested translation for function calls is given below.

```
e0(e1,...,en)
⟹
let f = e0 in
let a1 = e1 in
  ⋮
let an = en in
if is_class(f) then
    let o = create_object(f) in
    if has_attr(f, '__init__') then
        let ini = get_function(get_attr(f, '__init__')) in
        let _ = ini(o, a1,...,an) in
          o
    else o
else
    if is_bound_method(f) then
        get_function(f)(get_receiver(f), a1,...,an)
    else
        if is_unbound_method(f) then
            get_function(f)(a1,...,an)
        else
            f(a1,...,an)          # normal function call
```

Exercise 6.2. Extend your compiler to handle P_3. You do not need to implement operator overloading for objects or any of the special attributes or methods such as `__dict__`.

Appendix

6.4. x86 Instruction Reference

Table 1 lists some x86 instructions and what they do. Address offsets are given in bytes. The instruction arguments A, B, C can be immediate constants (such as $\$4$), registers (such as $\%eax$), 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
`addl` A, B	$A + B \to B$
`call` L	Pushes the return address and jumps to label L
`call` *A	Calls the function at the address A.
`cmpl` A, B	compare A and B and set flag
`je` L	If the flag is set to "equal", jump to label L
`jmp` L	Jump to label L
`leave`	`ebp` $\to$ `esp`; `popl %ebp`
`movl` A, B	$A \to B$
`movzbl` A, B	$A \to B$
	where A is a single-byte register, B is a four-byte register, and the extra bytes of B are set to zero
`negl` A	$-A \to A$
`notl` A	$\sim A \to A$ (bitwise complement)
`orl` A, B	$A \vert B \to B$ (bitwise-or)
`andl` A, B	$A \& B \to B$ (bitwise-and)
`popl` A	$*$`esp` $\to A$; `esp` $+ 4 \to$ `esp`
`pushl` A	`esp` $- 4 \to$ `esp`; $A \to *$`esp`
`ret`	Pops the return address and jumps to it
`sall` A, B	B << $A \to B$ (where A is a constant)
`sarl` A, B	B >> $A \to B$ (where A is a constant)
`sete` A	If the flag is set to "equal", then $1 \to A$, else $0 \to A$. A must be a single byte register.
`setne` A	If the flag is set to "not equal", then $1 \to A$, else $0 \to A$. A must be a single byte register.
`subl` A, B	$B - A \to B$

TABLE 1. Some x86 instructions. We write $A \to B$ to mean that the value of A is written into location B.

Bibliography

[1] V. K. Balakrishnan. *Introductory Discrete Mathematics*. Dover Publications, Incorporated, 1996.

[2] D. Beazley. PLY (Python Lex-Yacc). `http://www.dabeaz.com/ply/`.

[3] D. Brélaz. New methods to color the vertices of a graph. *Commun. ACM*, 22(4):251–256, 1979.

[4] P. Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Rice University, 1992.

[5] G. J. Chaitin. Register allocation & spilling via graph coloring. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, pages 98–105. ACM Press, 1982.

[6] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[7] A. H. Gebremedhin. *Parallel Graph Coloring*. PhD thesis, University of Bergen, 1999.

[8] S. Hack and G. Goos. Optimal register allocation for ssa-form programs in polynomial time. *Information Processing Letters*, 98(4):150 – 155, 2006.

[9] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture*, November 2006.

[10] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference, A-M*, November 2006.

[11] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2B: Instruction Set Reference, N-Z*, November 2006.

[12] S. C. Johnson. Yacc: Yet another compiler-compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, 1979.

[13] B. W. Kernighan and D. M. Ritchie. *The C programming language*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1988.

[14] M. E. Lesk and E. Schmidt. Lex - a lexical analyzer generator. Technical report, Bell Laboratories, July 1975.

[15] H. A. Omari, K. E. nbsp;Sabri Hussein A. Omari, K. E. nbsp;Sabri Hussein A. Omari, K. E. nbsp;Sabri Hussein A. Omari, and K. E. Sabri. New graph coloring algorithms. *Journal of Mathematics and Statistics*, 2(4), 2006.

[16] J. Palsberg. Register allocation via coloring of chordal graphs. In *CATS '07: Proceedings of the thirteenth Australasian symposium on Theory of computing*, pages 3–3, Darlinghurst, Australia, Australia, 2007. Australian Computer Society, Inc.

[17] K. H. Rosen. *Discrete Mathematics and Its Applications*. McGraw-Hill Higher Education, 2002.

[18] R. E. Tarjan and M. Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal on Computing*, 13(3):566–579, 1984.

[19] G. van Rossum. *Python Library Reference*. Python Software Foundation, 2.5 edition, September 2006.

[20] G. van Rossum. *Python Reference Manual*. Python Software Foundation, 2.5 edition, September 2006.

[21] G. van Rossum. *Python Tutorial*. Python Software Foundation, 2.5 edition, September 2006.