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.

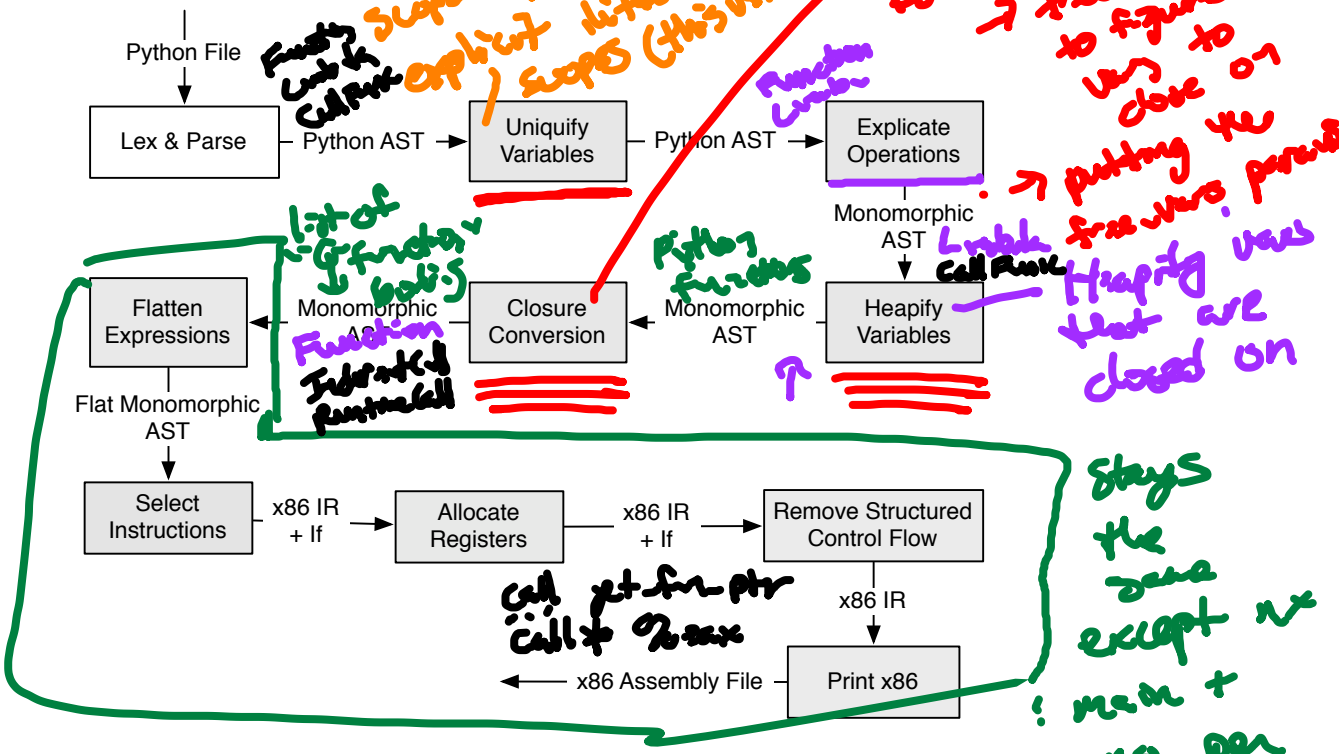


FIGURE 6. Organization of the compiler passes.

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` $f(\ldots)$: ...

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
```

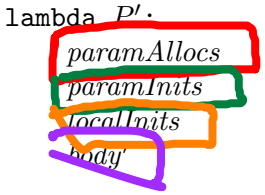
To convert lambdas to the new form, we simply put the body of the lambda in a return statement.

```
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:



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.