Assignment 7: functions and closure conversion

ECEN 4553 & 5013, CSCI 4555 & 5525
Prof. Jeremy G. Siek

October 20, 2007

The main ideas for this week are:

- first-class functions
- lexical scoping of variables
- closures and closure conversion

1 Syntax of P_4

This week we add functions to the language. We introduce two constructs for creating functions, the `def` statement and the `lambda` expression, and a new expression for calling a function with some arguments. For simplicity, we leave out function calls with keyword arguments. The concrete syntax of the P_4 subset of Python is shown in Figure 1.

```
expression ::= expression "(" [expression ("," expression)*] ")"
    | "lambda" [identifier ("," identifier)*] ":" expression
statement ::= "return" expression
    | "def" identifier "(" [identifier ("," identifier)*] ")" ":"
            statement
```

Figure 1: Concrete syntax for the P_4 subset of Python. (In addition to that of P_3.)

Figure 2 shows the additional Python classes for the P_4 AST.

2 Semantics of P_4

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 then called from several locations. The following is a simple example of defining and then calling a function in Python.

```
>>> def sum(l, n):
...     i = 0
...     s = 0
...     while i != n:
...         s = s + l[i]
...         i = i + 1
```

```
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_4 ASTs.

```
...     return s
>>> sum([1,2,3], 3)
6
>>> sum([4,5,6], 3)
15
```

The `def` statement creates a named function, whereas the `lambda` expression creates an anonymous function. 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.

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

Functions may be nested within one another as a consequence of how the grammar is defined. 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. Furthermore, functions are *first class* which means they are treated just like other objects: they may be passed as arguments to other functions, returned from functions, stored within lists, etc.. Figure 3 shows an example of a function that takes the derivative of another function.
A function may refer to parameters and variables in the surrounding scopes: which includes any parameters or local variables in enclosing functions and variables defined in the global scope. In the above example, the `lambda` refers to the `f` parameter and the `epsilon` local variable of the enclosing `derivative` function.

One of the trickier aspects of functions in Python is their interaction with variables. A function definition introduces a new scope, so a variable assignment within a function also declares that

```
>>> def derivative(f):
...     epsilon = 0.0001
...     return lambda(x): (f(x+epsilon) - f(x)) / epsilon
>>> def square(x):
...     return x * x
>>> square(10)
100
>>> ds = derivative(square)
>>> ds(10)
20.000099999890608
```

Figure 3: An example use of first-class functions. The `derivative` function takes a function as a parameter and returns a function.

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 2.1. Write five programs in the P_4 subset of Python that help you understand the language. Look for corner cases or unusual aspects of the language to test in your programs. Run the programs using the standard `python` interpreter. Add your test programs to the `test` directory.

3 Closure conversion

The major challenge in compiling Python functions to C functions is that functions may not be nested in C. (There is a GNU extension to enable nested functions, but we won't be using that extension.) When compiling to C, we must unravel the nesting and define each function in the global scope. 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 the global scope.

When we move a function, we have to worry about it's free variables. A variable is *free* with respect to a given expression or statement, let's call it P, if the variable is referenced anywhere inside P but not bound in P. A variable is *bound* with respect to a given expression or statement if there is a function or lambda inside P that has that variable as a parameter or local variable. 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 pair containing a pointer to the function and a list that stores the values of the free variables. This pair is called a *closure*. (We do not have support for pairs or tuples yet, so we'll use a list of length 2 to represent pairs.) When a closure is called, the list is passed as an extra argument to the function so that it can obtain the values of what use to be free variables from the list. Figure 5 shows the result of applying closure conversion to the derivative example. The `lambda` expression has been removed and the associated code placed in the `anonymous1` function. For each of the free variable of the `lambda` (`f` and `epsilon`) we add assignments inside the body of `anonymous1` to initialize those variables by subscripting into the `fvs` list. The lambda expression inside `derivative` has been replaced by a list of two elements: the first is the anonymous function and the second is a list containing the values of the free variables. Now when we call the `derivative` function we get back a closure. To

```
def free_vars_of_expr(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_of_expr(n.left) | free_vars_of_expr(n.right)
    elif isinstance(n, CallFunc):
        fv_args = [free_vars_of_expr(e) for e in n.args]
        return free_vars_of_expr(n.node) | reduce(lambda a, b: a | b, fv_args, Set([]))
    elif isinstance(n, Lambda):
        return free_vars_of_expr(n.code) - Set(n.argnames)
    ...
```

Figure 4: Computing the free variables of an expression.

invoke a closure, we call the function that is in the 0-slot of the closure and pass the 1-slot of the closure as the first argument with the normal arguments following.

```
>>> def anonymous1(fvs, x):
...     f = fvs[0]
...     epsilon = fvs[1]
...     return (f(x+epsilon) - f(x)) / epsilon
>>> def derivative(f):
...     epsilon = 0.0001
...     return [anonymous1, [f,epsilon]]
>>> ds = derivative(square)
>>> ds[0](ds[1], 10)
20.000099999890608
```

Figure 5: Closure conversion applied to the `derivative` function.

To implement closure conversion, write a couple recursive functions, `convert_closures_stmt` and `convert_closures_expr`. Both take one parameter: the current AST node. The two functions both return a new version of the current AST node and list of function definitions to be added to the global scope.

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

3.1 Function definition statements (`Function` nodes)

A function definition is converted into an assignment statement that assigns a closure to the name of the function.

`def` $funname$`(`$params, \ldots$`):`
  $body$
$\Longrightarrow$
$funname$ `= [`$globalfun$`, [`$fvs, \ldots$`]]`

The closure is a list (`List` AST node) with two elements, a mangled version of the function name *globalfun* that will identify the globally-defined version of the function, and a list whose

elements are initialized with the names of the free variables of the function. When the program executes, the variable references will be evaluate and the *values* of the variables will be copied into the new list. The function name must be mangled because there may be multiple functions with the same name inside child scopes which would conflict with each other when moved to the global scope. The notation $fvs, \ldots$ means the following. If $fvs = \{x, y, z\}$, then $fvs, \ldots$ is x, y, z.

The free variables of a function are the free variables FV of the body minus the local variables LV of the body and the names of the parameters. Note that a function defition nested inside the current one creates a local variable (the name of the nested function).

$$fvs = \mathrm{FV}(body) - \mathrm{LV}(body) - params$$

The case in `convert_closures_stmt` for function definitions returns the assignment statement defined above and a list containing the functions obtained by applying closure coversion to the *body* with the following function definition appended.

```
def globalfun(closparam, params,...):
  fvs_1 = closparam[0]
  fvs_2 = closparam[1]
  ...
  fvs_n = closparam[n - 1]
  newbody
```

where *newbody* is the result of applying closure coversion to the *body* and *closparam* is a freshly generated name. The function starts with a series of assignment statements, one for each of the variables in *fvs*. The right-hand side of the assignments should be subscript expressions that access the value for the free variable from the list passed into *closparam*.

3.2 Lambda expressions (`Lambda` nodes)

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
==>
[globalname, [fvs,...]]
```

where *globalname* is a freshly generated name and *fvs* is defined as follows, differing from function definitions in that we don't need to worry about local variables.

$$fvs = \mathrm{FV}(body) - params$$

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

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

3.3 Function calls (`CallFunc` nodes)

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 sould evaluate to a closure, i.e. a two element list with a function and array for the values of the free variables. We therefore need to transform the function call by calling the 0-index element of the closure and passing in the 1-index element of the closure as the first parameter, followed by the normal arguments. Because we need to use the closure resulting from e_f in two places, we assign the result to a temporary variable to avoid code duplication (and possibly duplicating side-effects).

e_f`(`e_1`, ..., `e_n`)`
$\Longrightarrow$
`{( pyobj f = `e_f`; f[0](f[1], `e_1`, ..., `e_n`) )}`

3.4 Heapifying variables

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

```
>>> x = 4
>>> f = lambda: x + 1
>>> x = 7
>>> f()
8
```

The read from variable `x` should be performed when the function is called, and at that time the value of the variable is 7. Unfortunately, if we simply copy the value of `x` into the closure's array, then the program would incorrectly output 2.

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. (We'll call this process heapification.) Consider what happens when we apply this idea to the above program. For convenience, we can use a single-element list to store the value of `x` on the heap. So we initialize `x` to `[4]` and then change all subsequent references to `x` (including assignments) to access the first element of the list: `x[0]`. The resulting program produces the same results as the original:

```
>>> x = [4]
>>> f = lambda(): x[0] + 1
>>> x[0] = 7
>>> f()
8
```

We can now perform closure conversion on this program without changing the result of the program.

```
>>> x = [4]
>>> def anonymous2(fvs):
        return fvs[0][0] + 1
>>> f = [anonymous2, [x]]
>>> x[0] = 7
>>> f[0](f[1])
8
```

Unfortunately, storing variables on the heap introduces extra run-time overhead because we have to read from memory, which is expensive on modern processors. Also, because our type analysis is not smart about lists, we lose the ability to give variables precise types. Thus, we only want to heapify the variables we really have to and leave the others alone. A conservative condition for when we need to heapify a variable is when it is a free variable of a function and is assigned more than once. For example, in the following program only variable `x` needs to be heapified. The variable `y` does not need to be heapified because it is only assigned to once and the variable `z` does not need to be heapified because it does not occur free in any functions.

```
x = 4
y = 2
z = 3.14159
f = lambda(): x + y
x = 7
z = 0.0
f()
```

Note that the decision regarding the heapification of variables need to be performed at every binding site for a variable. That includes global variables, function parameters, and local variables (which includes function definitions). For example, the parameter `x` and the local variable `y` of function `f` must be heapified in the following program.

```
>>> def f(x):
        y = 0
        g = lambda: x + y
        x = 3
        y = 2
        return g
>>> print f(1)()
5
```

The process of heapifying variables is best split into two passes. The first determines whether a variable needs to be heapified or name, and the second pass transforms the program.

3.4.1 Heapification analysis

The goal of this pass will be to attach some information to module, function, and lambda nodes that says which parameters or local variables need to be heapified.

To perform this analysis, write a recursive function over statements. The function will have three parameters, the current AST node, a set of variables that have been assigned to *after* the current statement, and a set of variables that need to be heapified. The recursive function attaches some information (heapify sets) to function nodes, returns the set of variables that have been assigned to in the current statement or after, and returns the set of variables that need to be heapified. The function is unusual in that it will process sequences of statements in reverse order, feeding the resulting set of assigned variables from each statement into the recursive call that processes the preceeding statement.

Let us look at how the important cases should be implemented. Let H_1 be the set of variables that need to be heapified according to statements after the current one and A_1 be the set of variables that are assigned after the current statement (both H_1 and A_1 are parameters passed into the analysis function).

For an assignment statement of the form x = e we first collect all the variables that occur free in any lambda's inside e. We then take the intersection of those with A_1, and union the result with H_1 to form H_2. We then return the sets $\{x\} \cup A_1$ and H_2.

Next consider a while loop:

```
while cond:
    body
```

The while loop is slighly tricky because any assignment in the loop must be considered to be "after" any of the variables that occur free inside functions or lambdas in the $body$. We therefore collect all variables that are assignment in the $body$ and take the intersection with all variables that occur free inside functions or lambdas in the $body$. We take the union of this result with H_1 to form H_2. We then take the union of the variables that are assignment in the $body$ (collected above) to the A_1 to obtain A_2. We return A_2 and H_2.

Handling function definitions is rather involved.

```
def funname(params, ...):
    body
```

Let $A_2 = A_1 - \mathrm{LV}(body) - params$ and $H_2 = H_1 - \mathrm{LV}(body) - params$. We perform heapification analysis on the $body$, passing in A_2 and H_2. This analysis gives us A_3 and H_3. We take the intersection of H_3 and $\mathrm{LV}(body)$ and attach the result to the current function node with the attribute name `local_heapify`. Similarly, we take the intersection of H_3 and $params$ and attach the result to the current function node with the attribute name `param_heapify`. We then return the sets

$$\{funname\} \cup A_1 \cup (A_3 - \mathrm{LV}(body) - params)$$

and

$$H_1 \cup (H_3 - \mathrm{LV}(body) - params).$$

3.4.2 Heapification transformation

Next we look how to transform the program to actually heapify the variables. The heapification is implemented as a recursive function with two parameters, the current AST node and a set of variables that need to be heapified. Let H_1 by this set.

Function definition statements We start by considering the case for function definitions.

```
def funname(params, ...):
    body
```

Let H^p be the set of variables in the `param_heapify` attribute and H^l be the set of variables in the `local_heapify` attribute. We start by recursively applying the heapification transformation to the $body$, passing in

$$(H_1 - \mathrm{LV}(body) - params) \cup H^l \cup H^p$$

as the set of variables to be heapified, and get back $newbody$.

We then transform the function definition by adding two series of assignment statements at the beginning, one series for the parameters that need to be heapified and one series for the locals that need to be heapified. For the parameters, we assign them to a one-element list that is initialized with the value of the actual parameter. For the locals, we assign them to a one-element list with a bogus element.

```
def funname(params,...):
    H_1^p = [H_1^p]
    ...
    H_n^p = [H_n^p]

    H_1^l = [False]
    ...
    H_n^l = [False]

    newbody
```

Variable expressions (`Name` nodes) If a variable `x` is in the set H_1 of variables to be heapified we need to replace the `Name` node with a `Subscript` node to access the element at index 0 of `x` (which now refers to a 1-element list due to the heapification process).

x
$\Longrightarrow$
x`[0]`

Lambda expressions (`Lambda` nodes) The transformation for lambda expressions is similar to that of function definitions, except that we don't have to worry about local variables. However, there is one wrinkle. We need to insert assignment statements to heapify some of the lambda's parameters, but the body of a lambda is suppose to be an expression, not a statement! In the closure conversion pass, which will come after heapification, we replace lambda's with functions, so it doesn't really matter if lambdas temporarily have statements in them. We therefore transform all lambda expressions to have a statement in the body. If the parameters require heapification, then there is a sequence of assignment statements at the beginning. In either case, the last statement in the body of the lambda will be a return statement with the result of heapifying the body.

```
lambda params,...: body
==>
lambda params,...:
    H_1^p = [H_1^p]
    ...
    H_n^p = [H_n^p]

    return newbody
```

4 Overview of the implementation

Figure 6 shows the suggested organization for the compiler.

Exercise 4.1. Update the `pyobj` union to include a member that is a function pointer.

Exercise 4.2. Update the function you wrote that figures out where variable declarations should be inserted to take into account function definitions and lambdas.

Exercise 4.3. Update the SSA conversion and type analysis to take functions into account. When analyzing function, assume the parameters have type `pyobj`. (We'll do something smarter next week.)

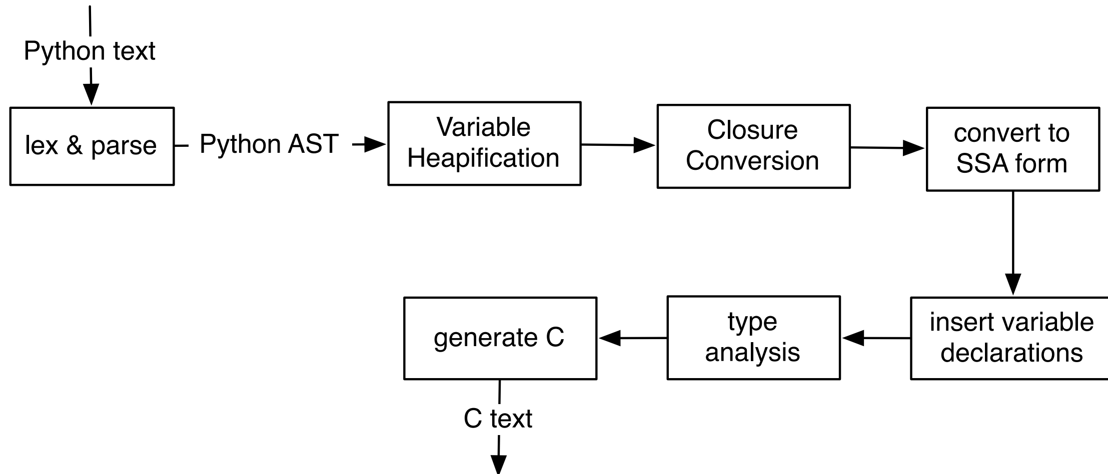

Figure 6: Organization of the compiler passes.

Exercise 4.4. Finish the function that determines the free variables of expressions and write a function that computes the free variables of a statement. Hint: for function statements (`defs`), make sure to subtract local variables from the set of free variables.

Exercise 4.5. (Turn this in.) Extend your Python to C compiler to handle P_4. Implement heapification and closure conversion as a separate functions that converts an AST to another AST. After closure conversion, the AST should not have any nested functions.