

Assignment 8: objects and classes

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

October 29, 2007

The main ideas for this week are:

- objects and classes
- attributes and methods
- inheritance

1 Syntax of P_5

The concrete syntax of P_5 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 ")" ] ":" statement
target ::= expression "." identifier
```

Figure 1: Concrete syntax for the P_5 subset of Python. (In addition to that of P_4 .)

2 Semantics of P_5

This week we add one new 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:

```
>>> 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
```

```

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

```

Figure 2: The Python classes for P_5 ASTs.

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

```

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):
...     pass

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

2.2 Objects

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

```
class C:
    pass

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 its 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, 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
47
```

You might wonder how does the Python implementation know 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 2.1. Read:

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

3 Compilation to C

Figure 3 shows the structure of the compiler with the addition of objects. We insert a new pass at the beginning of the compiler that lowers classes and objects to more primitive operations and

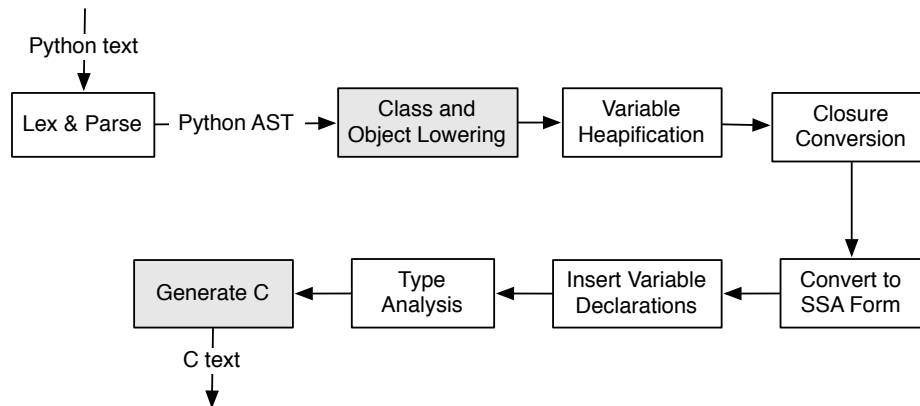


Figure 3: Structure of the compiler.

then we update the C code generation to handle these new primitives. The other passes will also need to be updated to pass along but otherwise ignore the new primitives.

In addition to the new passes and primitives, the entities introduced this week are all first-class, so the `pyobj` union needs to be extended:

class The runtime representation for a class should keep track of a list of base classes and a dictionary of attributes.

object The runtime representation for an object should keep track of 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 must include the function and the receiver object.

There are several operations on objects and classes that are best implemented as run-time C functions, such as looking up an attribute in a class hierarchy, and looking up an attribute in an object.

Compiling full class definitions is somewhat involved, so I first recommend compiling empty class definitions.

3.1 Compiling empty class definitions and class attributes

We begin with class definitions that have a trivial body.

```
>>> class C:
...     pass
```

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:
    pass
 $\implies$ 
C = AllocateClass()

```

where `AllocateClass` is a new AST node that is ignored by subsequent passes until we get to C code generation, where we output a call to a C helper function or macro.

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` should be translated into a `hashtable_search` in the class object's hashtable followed by an assignment to the value, if one was found, or else a `hashtable_insert` if none was found. This translation can be handled as you are generating C code.

The attribute access `C.x` in the `print` statement should be translated into a `hashtable_search` in the class object's hashtable. If the value of the attribute is a function, then one extra step is needed before returning the value: the function must be wrapped into unbound method. As above, the translation of attribute access can be handled as you are generating C code.

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:
    body
 $\implies$ 
C = AllocateClass()
newbody
```

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
 $\implies$ 
C.x = e'
```

where e' is a version of e where references to variables that are class attributes are changed into attribute access expressions, as described next.

Variables in expressions are translated to attribute access if the variable is one of the variables assigned in the body of the class.

```

x
 $\implies$ 
C.x
```

The translation for function definitions is:

```

def f(e1, ..., en):
    body
 $\implies$ 
def C_f(e1, ..., en):
    body # the body is unchanged, class attributes are not in scope here
C.f = C_f
```

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 side you won't know 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 rand(0,1) 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 describe the translation of function calls. For convenience we introduce two new AST nodes called `Let` and `SeqExpr` and use an alternative textual representation for the `IfExpr`. A `Let` expression assigns a value to a variable for the duration of a subexpression. I'll use the following textual representation for `Let`, where x is a variable and e_0 and e_1 are expressions.

```
let  $x = e_0$  in  $e_1$ 
```

A `Let` expression can be easily translated to C using the GNU statement expression feature.

```
{ pyobj  $x = e_0$ ;  $e_1$ ; }
```

The Python `IfExpr` 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  $e_0$  then  $e_1$  else  $e_2$ 
```

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

```
 $e_0(e_1, \dots, e_n)$ 
 $\implies$ 
let  $f = e_0$  in
if IsClass( $f$ ) then
    let  $o = \text{AllocateObj}(f)$  in
    if Hasattr( $f$ , '__init__') then
        let  $ini = \text{Getattr}(f, '__init__')$  in
        ({  $ini(o, e_1, \dots, e_n)$ ;  $o$ ; })
    else
        if  $n = 0$  then  $o$ 
        else Exit(-1)
else
     $f(e_1, \dots, e_n)$       # normal function call
```

I would suggest extending your AST with the nodes `IsClass`, `AllocateObj`, and `Hasattr`. In the following passes, such as closure conversion, etc., you can ignore these nodes, and then in the final conversion to C code you can produce the appropriate code for these nodes by calling C helper functions or macros.

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

```

The object contain a hashtable for their attributes. An assignment to an attribute associates the value of the right-hand-side with the attribute's name in the hashtable for the object. So the assignment `o.w = 42` inserts the item 'w':42 to `o`'s hashtable (or overwrite the value if there is already an item with a key equal to 'w').

To compile an access to an attribute of an object, first do a `hashtable_search` in the object's dictionary. If you find an item, return the value. Otherwise, go to the object's class (this should be stored inside the representation of the object) and do a search. (You already implemented search for classes in Section 3.1.) If the result is an unbound method, then create a bound method that contains the receiver object and the method. Otherwise return the result. This translation of attribute access can be performed as you are generating the C code.

3.4 Compiling bound and unbound method calls

Calls to bound and unbound methods show 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 check that the object is an instance of the class associated with the unbound method (when we add inheritance, objects of subclasses are OK too), and then call the underlying function. 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 and followed by the normal arguments. A suggested translation for function calls is given below. This translation uses several more primitive operations: `IsUnboundMethod`, `GetClass`, `IsBoundMethod`, `GetFunction`, and `GetObject`.

```

e0(e1, ..., en)
⇒
let f = e0 in
if IsClass(f) then
  let o = AllocateObj(f) in
  if Hasattr(f, '__init__') then
    let ini = Getattr(f, '__init__') in
    ({ ini(o, e1, ..., en); o; })
  else
    if n = 0 then o
    else Exit(-1)
else
  if IsUnboundMethod(f) then
    let o = e1 in
    if GetClass(f) == GetClass(o) then
      GetFunction(f)(o, e2, ..., en)
    else Exit(-1)
  else
    if IsBoundMethod(f) then
      GetFunction(f)(GetObject(f), e1, ..., en)
    else
      f(e1, ..., en)    # normal function call

```

3.5 Compiling inheritance

Compiling for inheritance only requires a few small changes to what we've already done. First, the `AllocateClass` operation should be extended to accept arguments that are the base classes of the new class. Second, attribute access for classes needs to be updated. Instead of just searching in the receiver class you must also search its base classes. The search should be performed in a depth-first left-to-right fashion. This can be accomplished by writing a recursive C helper function to perform the search.

Exercise 3.1. Extend your Python to C compiler to handle P_5 . You do not need to implement operator overloading for objects or any of the special attributes or methods such as `__dict__` and `__getattr__`.

Exercise 3.2. (Extra credit) Implement operator overloading, so for example, applying operator `==` to an object dispatches to its `__eq__` method.