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.

```c
/* 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.

```python
>>> 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

```c
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

```python
>>> 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.

```python
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!'
```

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:

```python
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!'
'
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(B1,...,Bn):
    body
    
body
    tmp = create_class([B1,...,Bn])
    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 fol-
lowing transformations. You will need to know which variables are
assigned to (which variables are class attributes), so before trans-
forming 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:

\[ x = e \]

\[ \rightarrow \]

\[ \text{set_attr(tmp, "x", } e' \text{)} \]

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.

\[ x = \]

\[ \rightarrow \]

\[ \text{get_attr(tmp, "x") if has_attr(tmp, "x") else } x \]
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.

\[ x = \text{get\_attr}(\text{tmp}, "x") \]

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

\[ x \]

The translation for function definitions is:

\[
\text{def } f(e_1, \ldots, e_n):
\]

\[
\begin{align*}
\text{body} \\
\text{def } f\_\text{tmp}(e_1, \ldots, e_n):
\end{align*}
\]

\[
\begin{align*}
\text{body} \\
\text{set\_attr}(\text{tmp}, "f", f\_\text{tmp})
\end{align*}
\]

### 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 \texttt{C} class.

\[ \text{C}() \]

In the AST, this is just represented as a function call (\texttt{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 \texttt{C} or it might call the function \texttt{foo}.

\[
\begin{align*}
\text{def } \text{foo}(): \\
\quad \text{print } 'hello world\n' \\
\end{align*}
\]

\[ \text{(C if input() else foo)}() \]

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 \texttt{__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 $e_1 \text{x if } e_0 \text{ else } e_2$ 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:

$$\begin{align*}
\text{if } e_0 \text{ then } e_1 \text{ else } e_2 
\end{align*}$$

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

$$e_0(e_1, \ldots, e_n) \implies \begin{cases}
\text{call } e_0 \to f \\
\text{let } f = e_0 \text{ in }
\text{let } a_1 = e_1 \text{ in }
\vdots \\
\text{let } a_n = e_n \text{ in }
\text{if } \text{is_class}(f) \text{ then }
\text{let } o = \text{create_object}(f) \text{ in }
\text{if } \text{has_attr}(f, '_\text{init}_\ldots') \text{ then }
\text{let } \text{ini} = \text{get_function}(	ext{get_attr}(f, '_\text{init}_\ldots')) \text{ in }
\text{let } _ = \text{ini}(o, a_1, \ldots, a_n) \text{ in }
on \\
\text{else } o \\
\text{else } f(a_1, \ldots, a_n) \quad \# \text{normal function call}
\end{cases}$$

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

```python
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:

\[
e_0(e_1, \ldots, e_n) \\
\quad \Rightarrow \\
\quad \text{let } f = e_0 \text{ in} \\
\quad \text{let } a_1 = e_1 \text{ in} \\
\quad \vdots \\
\quad \text{let } a_n = e_n \text{ in} \\
\quad \text{if is_class(f) then} \\
\quad \quad \text{let } o = \text{create_object(f) in} \\
\quad \quad \text{if has_attr(f, 'init') then} \\
\quad \quad \quad \text{let } \text{ini} = \text{get_function(get_attr(f, 'init')) in} \\
\quad \quad \quad \text{let } _ = \text{ini}(o, a_1, \ldots, a_n) \text{ in} \\
\quad \quad \quad \text{if is_bound_method(f) then} \\
\quad \quad \quad \quad \text{get_function(f)(get_receiver(f), a_1, \ldots, a_n)} \\
\quad \quad \quad \quad \text{else} \\
\quad \quad \quad \quad \text{if is_unbound_method(f) then} \\
\quad \quad \quad \quad \quad \text{get_function(f)(a_1, \ldots, a_n)} \\
\quad \quad \quad \quad \quad \text{else} \\
\quad \quad \quad \quad \quad \text{f(a_1, \ldots, a_n)} \quad \# \text{normal function call}
\]

Exercise 6.2. Extend your compiler to handle P3. You do not need to implement operator overloading for objects or any of the special attributes or methods such as \texttt{dict}. 