UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

CS 164 **P. N. Hilfinger**
Spring 2011

Project #3: Code Generation (version 5)

Due: Wed, 4 May 2011

The third project brings us to the last stage of the compiler, where we generate machine code. Beginning with the AST we produced in Project #2, you are to generate ia32 assembly code that will be assembled into a working program. We'll provide a skeleton containing a Project #2 solution.

When they are released, you will find a skeleton and supporting files in `~cs164/hw/proj3` and `~cs164/hw/lib` and in the `staff/proj3` and `staff/lib` subdirectories of the repository. We will include a parser and semantic analyzer that will provide trees properly annotated with declarations and types. You may also supply your own, if desired.

You can expect updates along the way (to make your life easier, one hopes), so be sure to consult the Project #3 entry on the homework web page from time to time, as well as the newsgroup, for details and new developments.

1 The Machine

We'll be using the ia32 architecture (as the family that includes the 32-bit Intel processors is called). We have provided you with an on-line, tutorial-style introduction (from Robert B. K. Dewar of NYU), and official Intel documentation. You can use the GCC compiler to look at what C code translates to:

```
gcc -S -g foo.c
```

which produces a file `foo.s`. Dewar's tutorial uses the Intel assembler format, which you can get GCC to produce with

```
gcc -S -g -masm=intel foo.c
```

and which you can have your project produce as well, by including the assembler directive

```
.intel_syntax noprefix
```

near the beginning of the assembly code you generate. (Unfortunately, Mac users, this option is not supported there.) The skeleton script will translate assembly-language programs you produce using a command like

```
g++ -o myprog -g myprog.s runtime.cc
```

This should work on the instructional machines (those using Intel architecture, that is), and on Intel-based GNU/Linux, MacOS X, and Cygnus installations. Furthermore, with this particular choice of options, the GNU debugger, GDB, will allow you to single-step through the assembly-language program while viewing the assembler source, and will also allow setting breakpoints and examining registers and variables.

Only some of the instructional servers use the ia32 architecture (rhombus, pentagon, cube, sphere, po, torus run i86pc Solaris). You can ssh into them as usual from home or from other instructional machines. You'll get error messages if you try to run your compiler on the wrong architecture.

2 Representation

We'll provide a runtime (in C++) that contains bodies for all the native methods in the standard library. This runtime implementation makes certain assumptions about the representation of Python values and objects. Python values will be pointers (references) to objects.

2.1 Variables

Variables (globals, locals, parameters, instance variables, and members of lists, dictionaries, and tuples) contain pointers to Python objects. In C++, we can represent this representation as follows:

```
typedef PyObject* PyValue;
```

2.2 Objects

A Python object may have one of several representations. All objects begin with a *type pointer* of type `PyType*`, described in §2.3. Here, as elsewhere, we'll use C++ notation to indicate the representations, even though valid C++ cannot precisely describe them in all cases. Since we are actually using assembly language, we needn't worry about this fact. We do not use C++ virtual methods, so none of these types has a C++ virtual table pointer; what you see is what you get.

Conceptually, we can think of an object as inheriting from the following structure:

```
typedef PyTypeObject* PyType;

struct PyObject {
    PyType type;     /* The dynamic type of this object. Also determines
                      * which arm of the union is applicable. */
};
```

Each subtype of `PyObject` will tack on a different set of trailing instance variables. The runtime will know which subtype to assume based on the value of the `type` field.

The builtin classes, by and large, have "native" representations specific to each, determined by the runtime system. Mostly, the runtime system we supply will deal with these

representations without your having to know about them, but there are few that you may find useful:

```
struct PyIntObject : public PyObject {
    int int_obj;
};

struct PyBoolObject : public PyObject {
    int bool_obj;   /* 0 for False, 1 for True. */
};

struct PyStrObject : public PyObject {
    int size;           /* Number of characters. */
    char data[size+1];  /* Contents. */
};

struct PyListObject : public PyObject {
    int size;           /* Current number of elements. */
    PyValue* data;      /* Pointer to values. */
    int capacity;       /* Current size of data array (>= size). */
};

struct PyTupleObject : public PyObject {
    int size;
    PyValue* data;
    /* Does not need a capacity, since size can't change */
};

struct PyXrangeObject : public PyObject {
    int lower_bound;
    int upper_bound;
};

/* ... etc. for other built-in types. */
```

User-defined classes (inheriting from type `object`) have the following representation:

```
struct PyClassObject : public PyObject {
    PyValue instanceVars[N];  /* Instance variables */
        /* N is determined from type. */
};
```

Function objects contain a code pointer and a static link, as indicated in lecture:

```
/* A generic function pointer (not really representable in C). */
typedef PyValue (*PyCodePtr) ();

struct PyFunction : public PyObject {
    PyCodePtr code;  /* A function body. */
    void* link;      /* A static link to pass when calling the
                      * body. This is the frame pointer of the function
                      * instance that created the function value. */
};
```

2.3 Types

A type is also an object (of type `type` in our Python dialect), although it has its own format.

```
struct PyTypeObject : public PyObject {
    int sig_size;    /* 0 for type objects of subtype PyDataTypeObject, or
                      * k>0 for PyFuncTypeObject with k-1 arguments. */
};

typedef struct PyDataTypeObject* PyDataType;

struct PyDataTypeObject : public PyTypeObject {
    PyDataType supertype; /* Supertype (0 for any). */
    const char* name;     /* Pointer to NUL-terminated string */
    PyValue exemplar;     /* Pointer to the (statically allocated)
                           * object of this type that contains the class
                           * variables.  See §6.6 */
    int ninstance;        /* Number of instance variables */
    int nmethods;         /* Number of methods. */
    PyCodePtr code[nmethods];  /* Addresses of methods */
};

struct PyFuncTypeObject : public PyTypeObject {
    int starred;        /* 1 if a star function, 0 otherwise. */
    PyType return_type;
    PyType formal_types[sig_size-1];
};
```

The runtime already contains type objects for all the builtin types. You need only build type objects for your own classes and for function types.

3 Scope of implementation

We'll provide a runtime that contains bodies for all the native methods in the standard library; type objects for all the builtin types and the values `None`, `True`, `False`, `stdin`, `stdout`, `stderr`,

and `argv`; and a number of runtime support routines to perform a number of tasks, such as type coercions.

Besides doing a lot in the runtime, we will reduce the implementation burden in a few ways

1. No garbage collection (we'll just let it pile up).
2. You may simply assume the usual Java modulo-2^{32} arithmetic for integers (yeah, it's a cop-out, but you do want to live to the end of the semester, don't you?).
3. Only downward function closures. You need not implement (or for that matter check for) function closures that survive return from the enclosing function. So for example, this need not work:

   ```
   def incr(k):
       return lambda x: x + k
   ```

   For a proper implementation, you'd also implement runtime checks to catch this error, but we won't bother doing that, either. In fact, you can feel free to put your function objects on the stack rather than the heap.
4. Don't worry about uninitialized variables. Simply initialize all variables blindly to 0.

4 The Runtime Library

The main program included in the skeletons will link the code you generate with our runtime system (written in C++), which provides:

- the main procedure, which initializes the runtime system and then calls the entry point to your code;
- type objects for the builtin types;
- functions for constructing dictionaries, lists, and tuples;
- functions for object creation, printing, type conversion, comparisons, and assorted other primitive operations called for by the semantics.

We'll maintain on-line documentation of the runtime system and of the runtime data structures used for functions, built-in types, and so forth. See the Project #3 entry on the homework page for a link to the runtime documentation (which is in the comments of the C++ source code of the runtime library).

5 What Your Compiler Must Do

The command

./apyc --phase=3 [-o OUTFILE.s] FILE.py

Compiles and the program in FILE.py, producing an assembly-language file `OUTFILE.s` (if defaulted, `FILE.s`). The output in this case will be in `gas` format (the GNU assembler). It should comprise the following:

1. Instructions that implement all of your functions, plus one special function for the main program. For the most part, these will look like the instructions produced for ordinary C functions, and you can use `gcc -S` to give yourself hints about what they should look like.
2. The virtual tables for all classes.
3. Declarations of global variables (those contained directly inside `__main__`.

./apyc -S [-o OUTFILE.s] FILE.py
A (traditional) synonym for the preceding command.

./apyc -c [-o OUTFILE.o] FILE.py
Compiles the program in FILE.py and assembles the result into OUTFILE (default FILE.o).

./apyc [-o EXECFILE] FILE.py
Compiles FILE.py and produces an executable in EXECFILE (default a.out).

There is one addtional option:

--library=DIR is as for the last project, and indicates the standard prelude, if it is not in the usual place. The directory also contains the file `runtime.cc`, containing the runtime library implementation.

All of this is set up for you in our project #3 skeleton files.

Your job is to write the code that takes the decorated tree (from project #2) and outputs assembly code for each of its modules. Your code must define two external symbols:

pyMain The label to your main program, which contains the concatenation of all the code that appears outside of functions or methods in the modules of your program (including code in class bodies that is outside any method.

pyMaxMethods A constant integer giving the maximum value of `nMethods` in all classes (including builtin ones). Our runtime initialization procedure uses this to fill in the `NoneType` class. We need it because our `None` value must be callable with *any* method (although the result will generally be to terminate the program, except for a few cases), so we must be prepared for any method index.

6 Code generation considerations

6.1 Type conversions, assignment, and parameter passing

An assignment

```
x = E
```

where `x` has type T and `E` has type T' will only be allowed by the initial phases of the compiler if T' is assignment compatible to T. If $T = T'$, all your code has to do is a simple copy. Otherwise, your code must call `pyCast` in the runtime library to check that the conversion is valid (see comments in the runtime library source for details). The same conversion is used to pass parameters, and to return values from functions.

6.2 Function prologues

Had we used the normal rules for subtypes of function types, we wouldn't need to do anything special to assign function values to variables, pass them as parameters, or return them. Unfortunately we allow cases such as

```
def double(x::int)::int:
    return x + x

x::(any)->any = double
print x("Hello, world! ")    # Should cause an error
```

Here, `double` is expecting an `int`, but, as shown here, could be called with a `str` (or `list` or `tuple`) through the function variable `x`. We have a similar problem with methods: one can override a method

```
def f(self, y):
    ...
```

in class `A` with

```
def f(self, y::int)::int:
    ...
```

in class `B`.

There are two strategies you can take:

1. Always have your functions and methods check the types of parameter values that are not `any` (using the runtime functions we provide for this purpose). This is probably easiest, though it is wasteful much of the time.

2. Have two entry points to each of your functions, one that checks types and one that does not. You can arrange them like this:

```
Entry1:
     check parameters
Entry2:
     push  ebp            Normal function prologue (Intel syntax)
     movl  ebp, esp
```

   Then use `Entry2` for normal function entry (when you call the function directly by the name used in its **def**) and the `Entry1` to form the code pointer in function values and in virtual tables (if there is a parameter of a type other than `any` in one of the parameters of the method that overrides a parameter of type in some supertype). This technique requires a bit more work to keep track of two labels.

6.3 For loops

Our subset will handle iterations much differently from full Python. A loop

```
for x in E:
    body
```

should be translated as if it were

```
_tmp0_ = E
_tmp1_ = _tmp0_.__len__()
_tmp2_ = 0
while _tmp2_ < _tmp1_:
    x = _tmp0_.__getseq__ (_tmp2_)
    _tmp_2 += 1
    body
```

where `_tmp0_`, `_tmp1_`, and `_tmp2_` denote compiler temporaries—local variables that are not used anywhere else.

6.4 Conditions

Python has a notion of *true values* and *false values,* which are relevant when implementing `if`. You can use the equivalent of the call `bool(`E`)` to get the truth-value equivalent of E. See §6.6 for how one calls `bool` and other allocators for builtin types. Alternatively, you can special-case instances where you know the static type (thus avoiding a method call).

6.5 Parameter lists with *

If a function has a trailing '`*`' parameter, it will expect to be passed a value of type `tuple`[1]. The caller will always know when it is calling such a function, and should silently create a tuple out of the trailing parameters.

6.6 Type exemplars and object creation

Consider the following code:

```
class A(object):
    x = 13
class B(A):
    y = 12
print A.x, B.x, B.y
```

In our dialect, the variables printed in the last statement are static variables that keep the initial values of the instances of `A` and `B`. That is, the allocation `B()` creates a new `B` object and initializes its instance variables from `A.x` and `B.y` (in our dialect, `A.x` and `B.x` refer to the same variable). Our runtime object-allocation routine will do this allocation for you if you set up your type objects for `A` and `B` properly. Each should contain a pointer to an *exemplar object* of its type, laid out just like an ordinary object of type `A` or `B`. The `x` field of `A`'s exemplar will contain `A.x` (and thus `B.x`) and the `y` field of `B`'s exemplar will contain

[1] a change from project #2, where it was type list; Python uses `tuple`, so we should do. If you use `list` instead, I suppose I won't mind too much).

`B.y`. The values of the other fields of these exemplars are irrelevant (they don't need type pointers, in particular).

So normally, an allocation, `A(x)`, where `A` denotes a type, should translate to the equivalent of

```
tmp = pyNew(A)   # pyNew is a runtime routine.
A.__init__(tmp, X)
```

and `tmp` holds the resulting value.

As a special kludge to mimic real Python, we'll handle allocations using builtin type names a little differently. A call such as `int(X)` should be translated as if it were

```
int.__init__(None, X)  # First parameter is ignored.
```

rather than the usual code. It is harmless to use the same code for both (`pyNew` does not allocation for builtin types), except that for normal allocation, the result is in `tmp`, and for builtin types, it is the result of `__init__`.

7 Code improvement

If you've been following carefully, you might notice that you need not do much of anything to implement all of Python's operators. Just about everything turns into a method call, so that once you get those working, our runtime library will supply most of the implementation. By making use of the compiler's knowledge of static types and of runtime reprsentations of some types (such as integers and lists), you can generate code that avoids these method calls and runs much faster. We do not *require* that you do this, but if you'd like extra credit, we'll give it to projects that run a few benchmark tests (simple loops involving integers) quickly enough.

You can get this speed by treating values of type `int` (or `bool`, although that is less critical) specially. Specifically, you can hold such values in "unboxed" form, taking them out of the objects (`PyIntObject`) that wrap them and then boxing them up again (using the appropriate runtime routine) when they must be passed to another function or returned.

You could do more as well, using knowledge of the structure of lists, tuples, and xranges, to speed up loops or fetches and assignments to lists and tuples. However, don't consider it unless you have time on your hands.

The insanely ambitious among you might consider doing real optimization—common-subexpression elimination, invariant code motion, constant folding, and the like. Again, we really don't recommend this, however, since you'll have more than enough to do as it is.

8 Output and Testing

For once, testing is going to be straightforward. Your test cases should be statically correct Python dialect programs (they may cause runtime errors, but they should get past the first two phases of the compiler). Testing should consist of making sure that the programs successfully compile, that they execute without crashing, and that they produce the correct output (or error out properly when there is a runtime error). As always, testing will be an important part of your grade.

9 What to turn in

You will be turning in four things:

- Source files.
- Testing subdirectories `tests/correct` and `tests/error` containing Python dialect source files and corresponding files with the correct output (for the `correct` subdirectory).
- A Makefile that provides (at least) these targets (make sure they actually work on the instructional machines):
  - The default target (built with a plain `gmake` command) should compile your program, producing an executable `apyc` program.
  - The command `gmake check` should run all your tests against your compiler and check the results.
  - The command `gmake clean` should remove all generatable files (like `.o` files and `apyc`) and all junk files (like Emacs backup files).
- A well-written design document describing the implementation, *including the parts we supply,* in sufficient detail to serve as a guide to a maintainer who wishes to fix bugs in or extend your compiler. More details on this will follow.

10 What We Supply

As usual, we'll have skeleton directories for you to start with in the staff repository. Take a look at the README file in `staff/proj3/README`. No doubt we'll be modifying a few things (again, in an attempt to make your life easier), so watch for on-line announcements

11 Assorted Advice

First, get to know the machine and assembly language by reading the documentation on the ia32 and experimenting with C programs on GCC. The problem in dealing with assembly language, of course, is that errors can have *really* obscure consequences. The GDB debugger can handle assembly language; its documentation is available through Emacs. The command `stepi` steps over a single instruction. You can use `p/i $pc` to print the instruction that is about to be executed; or use `display/i $pc` to set things up so that the next instruction is printed after each `stepi`. The debugger can display registers (with `p $eax`, for example).

You should definitely start writing lots of test programs, many of which you can test with Python (at least, after stripping type extensions).

Push through some simple stuff that does not involve classes. For example,

1. Be able to generate the main program for an empty source (all it does is exit without crashing).
2. Implement simple `print` statements for expressions.
3. Implement `if` and `while`.

4. Implement global variables and assignments to them.
5. Implement simple `def`s (with no `*` parameters) and calls.
6. Implement local variables.
7. Implement method calls (you can use them on builtin types without ever creating a class).
8. Implement classes with no methods.
9. Implement methods.
10. Add frills (`*` parameters, function values and types, etc.).

Again, be sure to ask us for advice rather than spend your own time getting frustrated over an impasse. By now, you should have your partners' phone numbers at least. Keep in regular contact.

Be sure you understand what we provide. Our software actually does quite a bit for you. Make sure you don't reinvent the wheel.

On the other hand: *feel free to modify anything!* The skeleton is *not* part of the spec. Modify however you see fit or ignore it entirely.

Keep your program neat at all times. Keep the formatting of your code correct at all times, and when you remove code, remove it; don't just comment it out. It's much easier to debug a readable program. Afraid that if you chop out code, you'll lose it and not be able to go back? That's what Subversion is for. Archive each new version when you get it to compile.

Write comments for classes and functions before you write bodies, if only to clarify your intent in your own mind. Keep comments up to date with changes. Remember that the idea is that one should be able to figure how to use a function from its comment, without needing to look at its body.