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

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

Project #3: Code Generation (Version 2)

Due: Wed, 7 December 2011

The third project brings us to the last stage of the compiler, where we generate executable code. Beginning with the AST we produced in Project #2, you are to generate C (or C++) code that will be further compiled into a working program, and you must fill in a runtime library to support the compiled programs. 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 in the `staff/proj3` subdirectory 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 Representation

In general, you will know the precise static types of things before using any operation that depends on this type. There are some unsound exceptions we discovered in Project #2 which, again, we are simply going to ignore (yeah, the program may crash; so what else is new?). As a result, for most operations (addition, subscripting, etc.), your compiler will know precisely what runtime functions to call, etc. Sometimes, you will be called on to assign things of unknown type (or pass them as parameters or return as values, all of which are forms of assignment). However, the idea is to use a representation for which all types have the same size, so that assignment need not know the type.

There is, however, one place where dynamic types are needed: the print routine. Therefore, your runtime representation must have a way of distinguishing the various types of items with some kind of tag in the objects (you can even use C++ virtual methods). While you can do this for the integer type as well—basically using the equivalent of Java's Integer type in place of primitive integer values—the limits on ints set in Project#1 allow a kludge. Since we only require support of 31-bit signed (and modify Python's semantics to do Java-style modular ("wrap-around") arithmetic, you can represent int values directly rather than as

pointers by using a simple (but actually standard) kludge. Represent an integer value N as $2 * N + 1$—that is, represent the actual value N in the uppermost 31 bits of a value, and mark this as an integer value by setting the least significant bit to 1. Normally, allocators for the ia32 architecture return word- or double-aligned pointer values (which always have 0 in the least significant bit), so this trick allows a general runtime routine to know when to treat something as an integer (doing an arithmetic right shift by 1 before operating) and when to treat it as a pointer. Alternatively, you can choose to set the last bit of *pointer* values to 1, but you'd better remember to compensate whenever you dereference.

2 Scope of implementation

You can make the following assumptions:

1. Garbage collection is optional (we'll just let data pile up). In fact, it is extra credit.
2. Again, you may simply assume modulo-2^{30} arithmetic for integers.
3. We're also going to treat the value None (represented by 0) as if it were 0 in arithmetic operations.
4. Don't worry about checking uninitialized variables. Simply initialize all variables blindly to 0.
5. Dictionaries need only handle bools, ints, strs, and tuples involving these types since other types are mutable and we don't have the necessary extension mechanisms for user classes.

3 What Your Compiler Must Do

The command

./apyc --phase=3 [-o OUTFILE.cc] FILE.py
: Compiles and the program in FILE.py, producing a C++ file `OUTFILE` (if defaulted, `FILE.cc`).

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

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

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

4 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.

5 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).

6 Assorted Advice

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

You can start immediately thinking about (and writing) the runtime module. Choose representations for the types (lists, strings, bools, files, dicts, tuples, ints). Identify the operations you will implement as calls to runtime routines and implement those runtime routines. The rules of Python require function closures. Therefore, you won't be able to simply translate local variables into C++ local variables; you'll have to put them somewhere. Figure out the structures you'll need for that.

For code generation, push through some simple stuff first. 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 literals (and the runtime routines).
3. Get integer expressions working.
4. Implement `if` and `while`.

5. Implement simple `def`s and calls.
6. Implement runtime library routines for lists, tuples, dicts.
7. Implement local variables.
8. Implement access to local variables of enclosing frames.
9. Implement classes.
10. *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.