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

CS 164 **P. N. Hilfinger**
Spring 2008

Project #3: Code Generation

Due: Friday, 9 May 2008

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.

You'll find skeleton and supporting files in `~cs164/hw/proj3` and in the `staff/proj3` subdirectory of the repository. We have included 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 webpage 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 online, 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`. Our `pythc` script will translate assembly-language programs you produce using a command like

```
gcc -o myprog -g myprog.s runtime.o
```

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, and ilinux[1-3].eecs run Fedora GNU/Linux). 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 The Runtime System

The `pythc` script included in the skeletons will link the code you generate with our runtime system (written in C), which provides:

- the main procedure,
- implementations of native methods,
- functions for constructing dictionaries, lists, and tuples,
- a function to create an object,
- functions for printing, type conversion, and assorted other primitive operations called for by the semantics,
- and the garbage collector.

On the instructional machines, we centrally maintain a compiled version of the runtime library, a compiler that performs parsing and static analysis, the standard prelude, and the file `sys.py`, containing the definitions in sys. When you work on the instructional machines, you won't need to (indeed should not) have copies of any of these; the `pythc` script will find them. At home, you'll need to check out these files as described in the README that comes with the Project #3 files in the staff repository.

We'll maintain online 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.

3 What Your Compiler Must Do

Your main program will read the tree produced by parsing and static semantics. This time, of course, the methods on the AST nodes will be concerned with code generation rather than static semantics, but otherwise things will be largely analogous to those in Project #2.

The output from your program will be assembly language 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. The exemplars for all classes (i.e., the static variables from which new instances are created). The built-in types have no instance variables or class variables, so each of these will consist only of a type pointer (virtual-table pointer).
4. Tables describing each class and each stack frame, for use by the garbage collector. Basically, these tell the runtime system where to find all the roots (see the garbage-collection lectures) and all pointer attributes of objects. Again, their format will be

documented in on-line notes. You don't write the garbage collector; it's part of the code we supply. Our garbage collector is not fully automatic, but only runs when called via a certain native method. That should slightly decrease the obscurity of the bugs caused by errors.

4 Optimization

There are a few opportunities for optimization relative to naive implementations of Pyth. We do not *require* that you do the clever thing, but we will be holding an execution-speed contest, and might even be persuaded to give a point or two to the fastest-running Pyth programs. Actually, it should require only modest effort to leave the standard Python implementation in the dust (on suitably chosen benchmarks).

You can't really do much except for things whose static types you know (and therefore whose representation you know). For example, if you know that something is an Int, there's a great deal you can do (since Pyth simply uses Java semantics for integers). For example, in the program

```
x: Int;  y: Int
x = 0; y = 0
while x < 1000:
   y += x; x += 1
```

the additions to `y` and `x` can be performed by `addl` and `incl` instructions. If you know that the controlling expression of a **for** loop is an Xrange, you could turn that loop into an ordinary C-style **for** loop.

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

5 Output and Testing

For once, testing is going to be straightforward. Your test cases should be statically correct Pyth 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. As always, testing will be an important part of your grade.

6 What to turn in

You will be turning in four things:

- Source files.
- A script file called `pythc` that glues together the pieces to produce a complete compiler. We'll supply a version of this script that should work for most of you, but you are free to modify it as needed. Please make sure it works on the instructional machines before turning it in.

- A testing subdirectory containing Pyth source files and corresponding files with the correct output.
- 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 program that your `pythc` script will run.
  - 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 `.class` or `.o` files) and all junk files (like Emacs backup files).

7 What We Supply

As usual, we have skeleton directories for you to start with in the staff repository. Choose either the java or c++ subdirectories to copy as your trunk. Also, take a look at the README file in `staff/proj3/README`.

You'll find a number of files in `~cs164/lib/proj3` on the instructional machines that your `pythc` will reference automatically. These include provided `pyc.parser` and `pyc.semant` packages, the standard prelude, the file `sys.py`, and the runtime library. These are also the repository under `staff/proj3/lib/proj3`. See the directions in the README file. No doubt we'll be modifying a few things (again, in an attempt to make your life easier), so watch for on-line announcements

8 Assorted Advice

What, you aren't finished yet? 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 has an interface very similar to GJDB (not accidentally); 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 Pyth test programs, many of which you can test with Python.

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.

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.

You *still* aren't finished?