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

CS 164 **P. N. Hilfinger**
Spring 2005

Project #3: Code Generation

Due: Wednesday, 4 May 2005

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 (with some additions and modifications by yours truly), you are to generate ia32 assembly code that will be assembled into a working program.

When the directory `~cs164/hw/proj3` becomes readable, you can copy the files we have placed there to help with this project. We will also update the `pyc.ast` and `pyc.semantics` packages to provide suitable decorated input for your code generator. Every significant identifier will have an appropriate Decl hung off it, and expressions will have Types.

Especially since this is the first time for the Pyth projects, 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 we will probably scrape together some additional reference material. You can the GCC compiler to look at what C code translates to:

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

which produces a file `foo.s`, and to compile and link assembly-language files:

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

Only some of the instructional servers use the ia32 architecture (examples: rhombus, pentagon, cube, sphere, po, torus). We will maintain a convenient online list. They all run Solaris, as does solar. You can ssh into them as usual from home or from other instructional machines. You'll get error messages if you try to run pythc on the wrong architecture.

2 The Runtime System

The `pythc` script that we'll maintain on the instructional machines 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.

We'll maintain online documentation of the runtime system and of the runtime data structures used for functions, built-in types, and so forth.

3 What Your Compiler Must Do

Your main program (in `pyc.codegen.Main`) will read the tree produced by parsing and static semantics. This time, the delegates of your 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 three things:

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. Since Pyth provides general closures, like Scheme, there is a little more to it than that, as will be documented in the online notes.
2. The virtual tables for all classes (the tables for built-in types are already in the runtime system).
3. 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). In particular, 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.

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 three things:

- Source files (in Java, since there seem to be no takers for C++).
- A testing subdirectory containing Pyth source files and corresponding files with the correct output.
- A Makefile that provides (at least) two targets:
  - The default target (built with a plain `gmake` command) should compile your program, producing an executable program called `pythc` (we will provide instructions for how to accomplish this in Java, so that you don't need the `java` command to run your program.
  - The command `gmake check` should run all your tests against your compiler and check the results.

7 What We Supply

We will shortly update the `pyc.ast` and `pyc.semantics` packages and the standard prelude for your use. We'll also add a few useful things to our own `pyc.codegen` package, as we did for Project #2. We'll be modifying a few things (again, in an attempt to make your life easier); watch the on-line documentation.

In addition, we'll provide you with a C library containing the runtime system, and a script that ties everything together (in past projects, we included this with the project files, but it will probably be safer to maintain it centrally this time). People at home will need GCC on an Intel machine (running Windows or Linux), as well as copies of all our files. Since this project will not run on MacOS X installations (sorry), I may just take the opportunity to convert the Java components to Java 1.5.

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. We'll try to make this a bit more convenient.

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 partner's phone number 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 CVS or PRCS is for. Archive each new version when you get it to compile. Either of these version-control systems will allow you to go back to earlier versions at will.

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.

> **New Policy:** Neither I nor the TAs will look at bug submissions for your code generator that produce debugging output (via printlns). Also, we'll expect you to convince us that you've made a credible effort to use the debugger before submitting a report.

You *still* aren't finished?