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

CS 164 **P. N. Hilfinger**
Spring 2009

Project #3: Code Generation (v2)

Due: Friday, 8 May 2008

The third project brings us to the last stage of the compiler, where we generate machine code. Beginning with a slight modification of the AST we produced in Project #2, you are to generate, in two stages, ia32 assembly code that will be assembled into a working program. We'll provide a skeleton containing a Project #2 solution and an interpreter whose language is a kind of three-address code. The problem is, first, to generate three-address code from the AST, and second, to augment the interpreter with code that will convert three-address code into ia32 assembler, suitable for input to gcc.

In order to make things more doable, we are ruthlessly pruning the language you have to implement. We will forgo implementation of many features that would have to be included in a production version, including many Python types, exceptions, and garbage collection.

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 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`. The skeleton 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 Modifications to Python

We're not going to implement the entire Python subset that the first two phases accept. Specifically, we will not expect you to implement the following (you may if you wish):

1. Floating-point numbers.
2. Several of the operators. The implementations of these operators are rather repetitive and unenlightening, so we'll just require a sample for the project.
3. Modules (import clauses).
4. Exceptions: **try** and **raise** statements.
5. **for** loops over things other than tuples and lists.
6. Slices.
7. Garbage collection.
8. Full integer semantics. You may simply assume the usual Java module-2^{32} arithmetic (yeah, it's a cop-out, but you do want to live to the end of the semester, don't you?).
9. Type-valued variables or expressions (for example, if `A` is a class, you need not worry about implementing `x = A`).
10. Initialization of objects other than by calling `__init__`. Basically, we're going to modify the front end to move all assignments to instance variables in the body of a class into the `__init__` procedure (creating one if needed). Thus, you will not have to worry about keeping around class variables that are distinct from instance variables.
11. Full 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.

Construct	Equivalent method call
`x + y`	`x.__add__(y)`
`x - y`	`x.__sub__(y)`
`x * y`	`x.__mul__(y)`
`x / y`	`x.__div__(y)`
`x ** y`	`x.__pow__(y)`
`-x`	`x.__neg__()`
`x[i]`	`x.__getitem__(i)`
`x[i] = v`	`x.__setitem__(i, v)`

Construct	Equivalent method call
`x < y`	`x.__lt__(y)`
`x > y`	`x.__gt__(y)`
`x <= y`	`x.__le__(y)`
`x >= y`	`x.__ge__(y)`
`x == y`	`x.__eq__(y)`
`x != y`	`x.__ne__(y)`

Table 1: Equivalent methods for Python operators. We differ from real Python in that we implement these methods for the basic types; normally they are just for extending user-defined types to allow use of standard operator syntax.

In addition, we've added a small piece of declarative syntax to the language: any parameter or local variable may be annotated to indicate its type. For example, any occurrence of a variable or parameter of the form "V`::Int`" will indicate that all occurrences of V must contain an integer. The only thing you are required to do with such variables is to generate a runtime check to make sure that any assignment to one of them is an integer, and (for parameters) to check that at the beginning of the containing function or method that they contain integers.

The standard prelude will define types `Int`, `List`, `Bool`, `Tuple`, `Dict`, and `String` as classes. If you consult the sections on Special Method Names in the *Python Language Reference,* you'll find that Python allows you to define operators such as `+`, `*`, array indexing, and many others my means of method calls. So, for example, in full Python, the expression `x+y` on user-defined types is equivalent to `x.__add__(y)`. See Table 1 for a summary of the methods we'll use (these aren't all of them; only the ones for the operators we require). This makes your task much easier, since we'll provide implementations of these methods for the builtin types as part of the runtime library we supply.

3 Representation

All our Python values will be *boxed,* meaning that all values are pointers. This includes integers, which will be represented similarly to Java's type java.lang.Integer. This uniform representation will allow a rather simple implementation in which all operations are method calls.

For method dispatching, we'll use virtual tables with the strategy called "Brute Force" from slide 19 of lecture 19. Every method or attribute name will have a unique table index distinct from all others, so that even when the compiler does not know anything about the type of `x` in `x.f()` (that is, `x` has type `any`), it still knows how to call `f`. As a result, many virtual tables will have many slots that stand for methods that are not implemented anywhere the class. These slots will be initialized to a special method that simply reports a fatal error and exits the program (since we don't have 'try' blocks).

There is a complication that arises from the fact that if `x` has type 'any', then an expression such as `x.y` might be either a reference to a method or to an instance variable. In full

Python implementations, this makes no difference, because such implementations simply use a dictionary for all attributes, be they methods or instance variables. But we're using an implementation that avoids the dictionary. So in fact, we'll use a table in which there are *two* entries for every method name. If `y` denotes a method in a particular class, then the first entry for `y` in that class's table will contain a pointer to the method and the second will contain 0. Otherwise, if `y` denotes an instance variable, the first entry will contain 0 and the second will be the offset of the instance variable. If the compiler happens to know the static type of `x`, it can figure out which entry to use without testing, and in the case of an attribute, it need not even use the table at all.

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,
- 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.

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.

5 What Your Compiler Must Do

Besides the original switches defined in previous projects, your compiler will also support the following command lines:

./apyc --run FILE.py
: Compiles and runs (interpretively) the program in FILE.py.

./apyc [-o OUTFILE] -S FILE.py
: Compiles the program in FILE.py and creates an assembly-language file in OUTFILE (default 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.

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

There are two options:

--prelude=PRELUDE.dast is as for the last project, and indicates the standard prelude, if it is not in the usual place.

--runlib=RUNLIB.c indicates the location of the runtime library, if it is not in its usual place. In real life, we would not keep this in C form, but we'll do it here to prevent certain "accidents."

6 Optimization

There are a few opportunities for optimization relative to naive implementations of Python. 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 compiled 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 we simply use Java semantics for integers). For example, in the program

```
x::Int = 0; y::Int = 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.

7 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. As always, testing will be an important part of your grade.

8 What to turn in

You will be turning in four things:

- Source files.

- A testing subdirectory containing Python dialect 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 `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).

9 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

10 Assorted Advice

What, you haven't started 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 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.

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* haven't started?