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

CS 164 **P. N. Hilfinger**
Spring 2010

Project #3: Code Generation

Due: Wed, 5 May 2010

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, 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 most features that would have to be included in a production version, including many Python types, exceptions, and garbage collection. The resulting dialect will essentially be statically typed, rather as if you were compiling Java.

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 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
```

near the beginning of the assembly code you generate. 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). 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. Perhaps the biggest change is that operands for all most binary operations are statically typed, so that an attempt to evaluate `x+y` or `x.a`, where `x` and `y` have static type Any, will be illegal. This should allow you (in principal) to generate efficient assembly code, similar to what you'd get in C.

We will further reduce the implementation burden as follows:

1. You need not implement any numbers other than integers.
2. You need only implement the operators listed in Table 1.
3. No modules (`import` clauses), except for those given in the standard prelude. Within those modules, only the methods or attributes given in the standard prelude.
4. No exceptions: no **try** and **raise** statements.
5. **for** loops only over tuples, lists, and xranges.
6. No garbage collection.
7. You may simply assume the usual Java module-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?).
8. No type- or module-valued variables or expressions (for example, if `A` is a class, you need not worry about implementing `x = A`).
9. 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:

Operation	**Operands**
$x + y$	Two integers, lists, tuples, strings
$x - y$, $x*y$ x/y, $x//y$ $x\%y$	Integers
x[v] D[y]	x a tuple, list, or string; v an integer. D a dictionary, y any type.

`is`
`and, or, not`
	All types.
	All types.

$x = y$, $x! = y$ $x < y, x > y, x \leq y, etc.$ x `in` L	All types Two integers or two strings only. L can be a dictionary, list, or tuple.
$x.a$	x an object with known class having attribute a, which may be a method only if $x.a$ is used as function in a method call.

Table 1: Operations you must support, and for what types.

```
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.

10. No user-defined operators (such as `__add__`).

11. Don't worry about uninitialized variables.

We've extended the declarative syntax of the language a little to include parameters, so that you can write

```
def f(x::Int):
    ...
```

This affects only the processing of variables in the body of the function, *not* the types of the formals as seen from outside the function (which are still usually 'any' except for some built-in procedures). Your program must cast (and check) assignments to typed variables and parameters and also must cast the values of typed parameters at the beginning of each procedure.

3 Representation

Although our language makes no distinction between "primitive" and "reference" types, unlike Java, our implementation of it will (invisibly to the user). Variables (or parameters) whose static type is `Any`, or any other type except `Int`, are represented as pointers to objects that contain (as suggested in lecture) a pointer to a virtual table that identifies their type and contains method pointers, if any. Variables of static type Int will be represented as 32-bit integers (the same size as pointers, conveniently). Naturally, this means that when assigning integers to variables of type `Any` (which include the elements of tuples, lists, and maps) or passing them to functions as parameters of static type `Any`, you will have to "box" them, creating objects analogous to those of type `Integer` in Java. Fortunately, we'll provide a runtime routine for this purpose.

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;
- functions for constructing dictionaries, lists, and tuples;
- a function to create an object;
- functions for 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.

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

We do not *require* that you produce optimal code, 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.

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.

You *still* haven't started?