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

CS 164 **P. N. Hilfinger**
Spring 2018

Project #3: Code Generation

Due: Fri, 4 May 2018

The third project brings us to the last stage of the compiler, where we generate executable code. Beginning with the AST you produced in Project #2, you are to generate C (or C++ or assembler) 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 the shared `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 alternatively supply your own, if desired.

You can expect updates along the way (to make your life easier, one hopes), so be sure to consult Piazza from time to time for details and new developments.

1 A Few Changes

We're taking advantage of the x86-64 architecture to make a couple of small changes in semantics.

- Extend integers to 32 bits (range -2^{31} .. $2^{31}-1$). We're also tweaking things so that the confusing business of representing the most negative number as one plus the most positive goes away.
- The integer arithmetic operations are to treat `None` as if it were 0 (so `1+None==1`, for example). The `is` operator can still distinguish 0 from `None`. This is just to make it possible to produce really fast code for integers for those of you who choose to do the optimizations described in §2.1.

2 Representation

We have provided a skeleton for a runtime support library, `lib/runtime.cc`. It contains a *suggestion* for representing types. You can change it at will (and in any case will need to complete it).

In general, you will know the precise static types of things before using any operation that depends on this type. As a result, for most operations (addition, subscripting, etc.), your compiler will know precisely what runtime functions to call, etc. In polymorphic functions, your program will have 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 values have the same size (because they are pointers), so that assignment need not know the type (it's just pointer assignment).

There is one place where dynamic types are necessary: the print routine. We allow programs such as

```
def f(x):
    print x

f(3)
f("Hi")
```

If there is to be one code sequence for `f`, the values passed to `x` will have to carry (dynamic) type information. Our skeleton cheats by using C++ virtual functions, and of course, you are welcome to do so too.

Don't worry about checking uninitialized variables. Simply initialize all variables blindly to 0.

2.1 Representing Integers

While you can represent integers as objects as well—basically using the equivalent of Java's Integer wrapper type in place of primitive integer values—the limits on ints set in Project#1 allow a kludge...er, that is, a faster representation. Since we only require support of a limited range of integers (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 convention. Store the actual integer value (rather than a pointer to an object containing that value) in the lower 32 bits of a 64-bit word, and set the upper 32 bits to a distinctive value that ensures that it differs from any possible pointer. As it happens, machines based on the x86-64 currently use only the lower 48 bits of their 64-bit pointers (for now). By setting the top 16 bits to something other than 0 (for example 0x8000), we get a value that cannot possibly be a valid pointer (and will cause a fault if you try to use it as such). When you send a value to a runtime function, it can recognize that value as an integer (if it needs to) by looking at those bits. Your code, knowing statically that something is an integer, can simply do arithmetic on the lower 32 bits without looking at the top bits. In fact, it need not even materialize those top bits until it wants to pass the value to a function.

For extra credit, you can implement this representation, and use direct integer arithmetic operations rather than function calls. That is, your compiler can "know" that a `binop` node such as

```
(binop 3 (id 3 __add__ 25) ... ...)
```

where 25 refers to the `Decl` for the definition in the standard prelude:

```
def __add__(x:: int, y::int)::int:
    native "__add__int__"
```

and handle this call specially.

2.2 Representing Closures

The other major representation decision you'll have to make is how to handle full closures, as used in Python. While `gcc` extends C (and C++) to allow nested functions and non-local variable access, these are not full closures, since one cannot validly return a nested function out of its containing function. C++11 introduces a form of lambda expression that "captures" local variables by reference or value, but these again will not quite get the desired semantics for Python.

For example, you might try to implement the Python function

```
def incr(n):
    """Return a vector of two functions, each of which increments
    an argument k by N+1."""
    def p(int k):
        return k + n
    v = []
    v.append(p);
    n += 1
    v.append(p);
    return v
w = incr(3)
print w[0](5), w[1](5) # Should print 9 (= 5 + (3 + 1)) twice
```

with the C++11 program

```
vector< function<int(int)> >
incr(int n)
{
    vector< function<int(int)> > v;
    function<int(int)> p = [n] (int k) -> int { return k + n;}; // Point (A)
    // The expression [n] .... is a C++ lambda that captures n.
    v.push_back (p);
    n += 1;
    v.push_back (p);
    return v;
}
auto w = incr(3);
cout << w[0](5) << " " << w[1](5) << endl;
```

It won't work, though: you'll get 8 printed twice rather than 9 because it is the value of `n` at Point (A) that is captured in `p`, and it does not track changes to `n`. Capturing a reference to `n` instead of `n` will (in general) result in garbage (since the variable `n` ceases to exist on return from `incr`).

Bottom line: you'll have to do some work!

3 Extras

There are a couple of things you can do for extra credit:

- Garbage collection is optional (we'll just let data pile up). We've put a function called `gc` into the standard prelude, but it need not do anything. You can have the function implement some form of garbage collection for extra credit. To test this, we'll give you some program that will fail if you don't collect garbage, along the lines of

```
for x in xrange(1000000):
    y = []
    for c in xrange(100):
        y[len(y):] = [x]
    gc()
```

- You can get extra credit if you successfully use the kludge we described (or something like it) to generate faster code for integers. We'll test this with a suitable timing test.

4 What Your Compiler Must Do

Besides the commands from previous projects, the skeleton provides two new ones:

`./apyc [ -o OUTFILE ] FILE.py`
: Compiles the program in FILE.py into a C++, C, or assembler file, and then compiles and links that program and the runtime support library `lib/runtime.cc` into an executable file `OUTFILE` (if defaulted, `FILE`).

`./apyc -S [ -o OUTFILE ] FILE.py`
: Compiles the program in FILE.py into C++, C, or assembler file OUTFILE (defaults to FILE.cc, FILE.c, or FILE.s as appropriate).

5 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. To indicate a runtime error, print a message with the word "error" in it (capitalized however you want) on the standard error output (called `cerr` in C++ and `stderr` in both C and C++). Testing will be an important part of your grade.

6 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). Both should contain semantically correct code (passing phase 2). The `error` directory should have programs that produce runtime errors.
- A Makefile that provides (at least) these targets (make sure they actually work on the instructional machines):
  - The default target (built with a plain `make` command) should compile your program, producing an executable `apyc` program.
  - The command `make check` should run all your tests against your compiler and check the results.
  - The command `make  clean` should remove all generatable files (like `.o` files and `apyc`) and all junk files (like Emacs backup files).

7 Using Git to Get Started and to Submit

If you have set up your local copy of your GIT team repository as in our setup directions, one member of your team can start your project from this point with the command:

```
$ cd team-repo
$ git fetch shared
$ git merge proj3 shared/proj3
```

(As usual, first be sure that you have first committed any changes in the current working directory.) This merges a `proj3` subdirectory into your master branch. You can also put it in a separate branch.

Should we make modifications to our skeleton, you can incorporate them into your files (if desired) with the following steps:

```
$ git commit -a    # If needed.
$ git fetch shared
$ git merge shared/proj3  # To incorporate get changes to proj3 skeleton
```

We will test your program by first translating a suite of semantically correct Python programs (checking that your program exits cleanly with an exit code of 0), and checking that the translations produce the proper results. There will also be a suite of programs with runtime errors, which should compile successfully and then produce proper error messages when run, exiting cleanly with an exit code of 1.

Not only must your program work, but it must also be well documented internally. At the very least, we want to see *useful and informative* comments on each method you introduce and each class.

8 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, dicts, tuples, ints, ranges). 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.*

The skeleton suggests a single code-generation procedure, `codeGen`, but this is just filler. You may want to replace it with a richer method (or methods). In any case, you'll need more methods to handle such issues as allocating local variables.

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. 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.