Building an Interpreter With RPython

Release 0.1.0

Aug 31, 2017

Contents

This repository contains introductory material on building a simple interpreter with RPython.

The original slides are also available.

Contents:

CHAPTER 1

PyPy

Besides being the most prominent example of an interpreter written in RPython, having been the interpreter for which RPython was developed, we will also use PyPy in order to actually translate our own interpreter, which we will discuss later.

Below are brief instructions for installing and running PyPy.

Installing PyPy

PyPy can be installed by following the instructions at:

http://pypy.org/download.html

or:

OS X

```
$ brew install pypy
```

Windows

Windows binaries are available at the above link.

Linux

PyPy should be available in your Linux distro's package manager.

Alternatively, tarballs are available at:

https://github.com/squeaky-pl/portable-pypy

which should work across distributions.

Running PyPy

As with CPython, the REPL is accessible via `pypy` at a terminal.

Feel free to play around a bit if you've never done so already, whereby you'll (hopefully!) notice nothing amiss.

CHAPTER 2

CyCy

CyCy is a small C interpreter built during a 48 hour hackathon at Magnetic.

It certainly does not interpret an appreciable amount of C, nor does it run particularly quickly, but it's a small step beyond the example interpreter (which we'll also see soon).

It is highly recommended to resist temptation on directly copying its layout (or the layout of the example interpreter) line for line. Reading it briefly and using it as a reference when stuck will allow you to proceed without (much) frustration while still taking time to play with RPython enough to experience it.

Installing CyCy

Clone the CyCy repo which can be found at https://github.com/Magnetic/CyCy:

```
$ git clone https://github.com/Magnetic/cycy
$ cd cycy
```

Optional: Create a Virtualenv

Create a virtualenv which you will install CyCy into. If you have no other preference on location, create one within the CyCy repo checkout:

```
$ python -m pip install -U virtualenv
$ virtualenv -p pypy venv
```

which will create a virtualenv (a directory) named `venv` in the checkout, and will use the PyPy interpreter you've installed inside it.

Install CyCy as you would a normal Python package:

```
$ venv/bin/pip install -e .
```

which should fetch all of the necessary dependencies.

Run `venv/bin/rpython --help` to confirm that the toolchain was successfully installed, and you're on your way.

Alternate: Without a Virtualenv

If you do not wish to use a virtualenv, you can install CyCy and its dependencies globally:

```
$ pip install --user -e .
```

Follow along with the rest of the document by removing `venv/bin/` from the beginning of commands, since your binaries will be available globally.

Note: If after running the above command you cannot run `rpython --help`, you likely do not have `~/.local/bin` on your shell's `$PATH`. Follow the instructions of your shell on adding it (typically either to `.bash_profile` or `.zshenv` if you're using bash or zsh respectively).

Running & Translating CyCy

Take a quick look at the contents of the directories in front of you. We will cover the types of components in them as we develop our own interpreter, but it will be useful (and perhaps somewhat irresistible) to peek at what reasonably small amount of code is there.

At a high level, the translation process will take the code you have and translate it into an executable you can run standalone.

Before we do so, let's prove that the interpreter can simply be run as a normal program *on top* of Python. Running:

```
$ venv/bin/pypy -m cycy
```

should present you with a CyCy REPL. Here you certainly *will* notice things amiss (bonus points if you simply crash the interpreter).

But! If you run something simple, like:

```
CC-> int main (void) { return 2 + 23; }
```

you should see:

```
25
```

outputted below. There are various slightly more complicated things for which CyCy manages to implement support for so far.

Note: For both CyCy and the interpreter we will attempt to build, you may find it helpful to install the `rlwrap` *nix utility and to use it as a wrapper around the REPL:

```
$ rlwrap venv/bin/pypy -m cycy
```

will give you a REPL with readline support provided via rlwrap.

You can find it in homebrew or via your Linux package manager.

Now that we can run CyCy on top of Python, let's use the RPython toolchain to create an executable that is *independent* of the Python runtime.

From the same directory (the root of the checkout)::

```
$ venv/bin/rpython --output=cycy-nojit cycy/target.py
```

will produce a long stream of output that you might find interesting to stare out.

After around 2 minutes of churning, out should pop an executable in the current directory which you can run via:

```
$ ./cycy-nojit
```

which should produce a REPL with (*roughly*) the same behavior as before.

This executable depends neither on Python nor the RPython toolchain:

```
$ file ./cycy-nojit
$ otool -L ./cycy-nojit
```

which should show you some basic information on the executable (and specifically that it does not in fact try to link against Python). On Linux, use `ldd` in place of `otool -L`.

CHAPTER 3

Some Example Interpreters

As we move forward with our own interpreter, it will be helpful to have other interpreters that we can reference if and when we are stuck, either with design or implementation.

To start with, besides CyCy which we have already cloned, let's retrieve two more interpreters written with RPython.

First, let's clone the "official" RPython example interpreter. It lives on BitBucket.

Note: To clone it, you'll need the mercurial VCS. If you do not already have mercurial, you can install it as any other Python package via

$ pip install –user mercurial

In this instance personal recommendation is to install mercurial globally, but feel free to install it into a virtualenv as well if you wish.

Execute:

```
$ hg clone https://bitbucket.org/pypy/example-interpreter
```

to grab a copy of it. Take the same quick flip through the source code as you did through CyCy. You'll hopefully notice some surface-level similarities because CyCy was written via the same strategy of occasional glancing at the example interpreter for inspiration.

As we progress through writing our own interpreter, you now have at least a pair of interpreters to reference. In order of complexity, we have:

- the example interpreter implementing a toy language similar to our own
- CyCy implementing a small subset of C

Let's also clone the Topaz and PyPy interpreters as well. These interpreters are full-blown (real-world) projects with all of the considerations that brings, so they come with huge additional levels of complexity. Try not to be flustered by it, we clone them now for two reasons:

- familiarity with Python might in some way provide additional guidance if you can manage to find the associated implementation in PyPy

- both repos are large – we might want to take quick looks at PyPy later in the day, so we'll clone them up front. Feel free to leave the clone running in the background while we proceed.

You can find Topaz at https://github.com/topazproject/topaz

CHAPTER 4

Other Resources

There are a number of other resources that will be useful as we work:

- the RPython documentation which you'll likely want to keep open for reference throughout the tutorial
- the #pypy IRC channel on Freenode which you might like to join, especially if you have an issue that we can't figure out in person. See http://webchat.freenode.net/ to connect if you do not already have an IRC client.

CHAPTER 5

RPython

There is an introduction to RPython in the RPython language documentation, which explains what subset of Python constitutes valid RPython.

It is *not* critical that you read that page from top to bottom (yet, probably not even at all at least for today).

The most important things to remember from the start are:

- You cannot mix objects of "un-unifyable" types in the same collection. Lists must contain all strs, or all instances of a class. It does not have to be the same exact class, the classes just have to be unifyable, which means that there is some common base class between them (at the RPython level this cannot be `object`, so you can't mix ints with instances of your RPython class). Relatedly, you can mix `None` with some types, but not others. See the documentation page above for details.
- Most of the (Python) standard library (and similarly external modules) are not valid RPython. There is a separate, smaller RPython standard library. With a few exceptions, it comprises the external RPython code you have available.
- Write simple code and it is likely to be easily converted to RPython even if it is invalid. If your code is complex, it will likely need untangling once you try and translate it.

Unfortunately the most important rule though is the one at the top of the documentation, that if it translates, it's valid, and if it ain't, it isn't.

Two Specific Notes

Besides the above, there are two specific things which you might find surprising initially that are worth learning upfront.

Firstly, the RPython toolchain will expect an entry point similar to the one expected when writing C code. By default it will expect a function called `main` in the file you pass to the `rpython` binary (explained shortly). The function should return an `int` as `main` would in C. This is where the translation process which we'll discuss (as well as your interpreter) will begin.

Secondly, the `assert` statement in RPython has special semantics. Whereas in Python it is generally used to describe invariants that should never be false in a piece of code, in RPython it *additionally* makes an assertion of that invariant *to the toolchain itself* so that the toolchain can make use of that information.

As a quick example, you'll likely soon notice once we begin writing our parser that the toolchain will complain during translation if for instance you have:

```
class Parent(object):
    ...

class ChildA(Parent):
    attr_only_on_this_child = 12

class ChildB(Parent):
    ...
```

and you try and access `attr_only_on_this_child` even if you only pass in instances of `ChildA`. You will need to tell the toolchain that you are *guaranteeing* that only instances of `ChildA` will be there, and not `ChildB`, and the way to do so is by saying `assert isinstance(myinstance, ChildA)`, which signals that exact fact.

CHAPTER 6

A Start

The "language" we'll implement is exceedingly simple. We'll aim for a small initial goal:

- variable assignment
- string and integer literals
- simple output to stdout

We'll call our interpreter (and language) Snap. Don't bother Googling it.

The simple syntax for our language will lead us to aim directly for the following small program:

```
foo = 2 + 7
bar = "cat" + "dog"
print(foo)
print(bar)
```

where we will want:

```
$ snap example.snap
```

containing the above program to put the appropriate output on stdout.

We'll tackle our interpreter in 3 phases, a small parser, a small corresponding bytecode compiler, and then a small corresponding bytecode interpreter.

After that we'll embellish!

Initial Setup

Fork the repo containing these instructions, which you can use to house your interpreter (feel free to just use the local clone if you prefer). You can use the CyCy repo as a guide (or any of the other interpreters) for all of the basic steps below. We will assume a similar layout for the remainder of this session.

It lives at https://github.com/Julian/BuildingAnInterpreter.

We'll be creating a fairly typical Python package, despite its purpose, so create the package layout you are comfortable with, along with a virtualenv for your project (be sure to use `-p pypy` if you do). Install the `rpython` toolchain which we'll shortly need from PyPI.

We'll be writing tests! Install your favorite test runner as well, and follow along with the next step, where we finally start writing our parser.

CHAPTER 7

A Snap Parser

Our first goal is to be able to successfully parse our goal into an AST. An AST is a programmatic representation of the structure of our program (which you may have encountered via the `ast` module in Python).

Create a test file named `test_example.py` in your Snap package's test folder, and create a test case within it. The test case will guide us along the way towards at least being able to parse our example program. We'll handle interpreting it soon.

We wish to parse the example program we had earlier:

```
foo = 2 + 7
bar = "cat" + "dog"
print(foo)
print(bar)
```

into an AST. There are numerous ways to represent the above source code as an AST. Feel free to be a bit creative, but as a sampling, we will represent the first sample of this piece of code as the following (RPython) AST:

```
Assign("foo", BinaryOperation("+", Integer(2), Integer(7)))
```

See if you can write a (failing) test case that asserts that the result of parsing the above source should produce that AST. You'll need to create a few classes corresponding to the various node types in your AST.

Note: Hooray! Tests don't need to be valid RPython, they won't run within your interpreter, so you have freedom to write them however you'd like.

We'll follow roughly typical TDD (if you're unfamiliar, don't worry too much about it), so create the function that will parse your AST and make your example test pass by simply giving it a fake implementation that simply returns the (overall) AST we want for our example program.

We'll now build out our parser in small chunks until we successfully can parse the above program.

RPly and an Actual Start to Our Parser

We need a way to actually parse our source code into the AST we've just designed. There are a number of paths forward. The most obvious is to implement a parser ourselves "by hand", but there are a number of tools that exist to make this easier. The RPython standard library has the `rpython.rlib.parsing` module, but we'll use the RPly library, which has a slightly nicer API.

There are only two pages of documentation, so have a quick read through them to see how you will likely want to proceed. You'll likely find consulting the CyCy parser helpful minutes as well but try and wait until you're off the ground.

You may want to consult additional resources on EBNF notation if this is your first exposure to it. For our purposes superficial knowledge should suffice until you want to extend the parser (later) as well, so it can also wait.

Break down the things you will need to parse into smaller chunks, write unit tests for the (small AST) that will result from parsing them, and then implement enough of a parser to make your tests pass, extending your implementation at each step. For example, you might want to take the following path:

- Parse integer literals
- Parse sums of integers
- Parse assignment of an integer
- Parse the assignment of a sum
- ...

slowly building up your parser via your unit tests.

Translation

Let's not forget about translation! Generally speaking, you should *not* necessarily attempt to translate your code not after change or commit (it's tedious and long as you'll soon notice). Rely on your tests to check that your code works, then simply periodically attempt to translate, and fix any invalid RPython you've introduced since the last time.

Let's set up enough to be able to translate our project and check if it's valid RPython. Recall that translation only operates on code paths that are actually reachable from your entry point, which we're about to define, and only on objects *after* import-time, where you're able to fully utilize Python.

Note: If you forgot about the type unification requirement of RPython, you'll likely see your first translation error with your AST nodes!

These errors are cryptic, but if you stare at them and trust them for long enough, they often are clear enough to point you in the right direction.

See if you can solve your error.

More About non-RPython

With the above note again in mind about translation operating only on code paths your interpreter will traverse (and not ones used only during tests or debugging), it's useful to set yourself up for easy debugging of your parser.

Give your AST nodes a helpful `__repr__` and see what you can do in regular Python to help yourself as you go along.

You've likely already implemented `__eq__` and `__ne__` methods on your nodes to get your tests to pass. These too do not need to be valid RPython since they likely will only be used in tests.

A Look Ahead

In a coming exercise we'll begin writing a simple REPL, similar to the Python REPL (or CyCy REPL).

You might want to look ahead and implement a simple REPL that simply can display the AST for the inputted source code, which may be helpful as you move forward.

CHAPTER 8

A Snap Compiler

We have the foundation of a parser, which means we now can convert source code into programmatic representations of an AST.

Our objective now is to be able to compile an AST into bytecode. Our bytecode, roughly speaking, is similar to bytecode you likely will have encountered at least in passing with Python.

It is a sequence of bytes which program (what will become) our virtual machine, in an analogous way to how machine code is a sequence of bytes that programs our CPU.

Our compiler will take an AST and walk it to produce a single stream of bytecode that encapsulates the instructions we need to execute.

Designing bytecode languages is an art unto itself, but our fundamental goal during compilation is to create a simple language that manipulates a stack which we will maintain as our VM.

For example, here is a human-readable rendering of what adding two numbers might look like for our VM:

```
LOAD_CONST 0
LOAD_CONST 1
BINARY_ADD
```

The interpretation of these three instructions are:

- load a particular constant, whose ID is 0 (we'll maintain constants in an array, so this will be an index specifying the constant within the array), then push its value onto our stack
- do the same for another constant with ID 1
- pop the top two entries off the stack, perform addition, and push the result back onto the stack

The non-human readable version should use simple bytes to represent the actual instructions ultimately, which the VM will then dispatch on.

First Steps

Your compiler should operate on an AST, and ultimately will produce bytecode: a simple sequence of bytes which our upcoming VM will execute.

In fairly typical OO style, generally it will simply delegate to each of your AST nodes for compilation, in which case each node will know how to compile itself by producing a sequence of our bytecode that it corresponds to.

We'll need to store state during our compilation process. We need at least a place to store variables and one to store constants that the bytecode we generate will reference.

For a concrete example, a simple constant node like the one we'll need for integers or strings should compile itself by simply registering itself as a constant on an object that is keeping context for the compilation. It should then emit a bytecode that when executed will retrieve the corresponding constant from the context object.

In the bytecode we will emit, loading this constant for use on the stack will consist of accessing an appropriate element within a registered constants list by index.

Compiler Tests

Write your first compiler test! There are (at least) two ways to test your compiler. One is in integration with your parser, by asserting that some inputted source code parses and then compiles into an expected piece of bytecode. The other is to test direct compilation of an AST.

Try out both methods and see which you find easier to read. You may also find it convenient to have a way to produce human-readable bytecode dumping for your bytecode, which should produce output similar to our first example above, or to what you'd get out of the dis module.

CHAPTER 9

An Interpreter

Finally! It's time to actually execute some code.

We have bytecode, which essentially is a tape-like sequence of instructions that we will interpret. We've casted our (potentially) complicated language into a sequence of simple stack manipulation operations.

Some of you will find interpretation to be the most "fun" part of the process. We need to implement the appropriate stack manipulation for each bytecode we wish to interpret.

The Main Loop

Interpreting bytecode will take place within a main loop similar to the CPython VM's main loop or PyPy's main loop. A giant loop that simply performs whatever bytecode instruction is at the current position we're processing.

A program counter should keep track of that position within our bytecode. Our interpreter will read sequentially through the bytecode sequence, possibly moving the program counter to some other position (if you implement a jump or conditional expression in a later exercise).

Along with a stack (an RPython list), we'll implement our plan.

For each bytecode instruction that your compiler produces, implement the appropriate stack manipulation.

Note: Depending on the particular bytecode instruction, you may find it difficult at this stage to write tests without simply making tests that assert about the internal state of your stack.

Try this out.

You might find it more reasonable as you progress to write tests that use the print bytecode mentioned below instead once you have a working entry point.

Print

Until you implement full support for function calls, it will likely be useful to special-case the print() instruction by giving it its own bytecode.

Note that you cannot use the `sys` module in RPython for the most part, nor do you have access to `open`. You may write to stdout directly via `os.write` by passing in an fd of `1` for stdout.

You may also want to check out the `streamio` module from the RPython standard library which can provide some provisional file-like support for RPython.

Once you have the ability to print values, you can begin print-debugging your own interpreter!

Wrapper Objects

Much like Python, our toy Snap language allows you to print objects that aren't necessarily strings.

It becomes useful to start applying OOP techniques to objects at your *language level* and not just at the RPython level for your interpreter. For example, you may have an integer object which represents a Snap integer within your runtime.

The convention is to call objects like these `W_Integer`, for example, where the `W_` prefix indicates that this object is a wrapper object.

Once you have wrapper objects, you can begin to encapsulate Snap features on each wrapper object. A `W_Integer` for example may have a `.to_string` method in RPython, which returns a `W_String` Snap string. Your `PRINT` bytecode might then be implemented by simply delegating to this method in order to produce a string, which you can implement polymorphically on each Snap type you might have.

If you begin to implement method calls in Snap, your `to_string` method might further become a Snap-accessible method that you can call on your Snap objects directly if so chosen.

CHAPTER 10

A REPL

As a somewhat fun additional exercise, let's implement an interactive interpreter which parses, compiles and executes your code in the same way the Python interpreter does.

Most of the work here is straightforward.

You do not have access to the `code` (Python) stdlib module, so it will be up to you to manually implement a simple execution loop.

Simply repeatedly read a line, and run it through the appropriate steps within your interpreter.

Here specifically you might want to take a look at the `streamio` module if you did not before.

Final Thoughts

To come full circle, verify that you can run the example that we set out to interpret.

It should produce our expected output.

CHAPTER 11

Going Further

At this point you likely have a number of further ideas for extension.

Implement them!

For some further inspiration:

- Loops! Function calls! Methods! More operators. Extend your parser, compiler and interpreter to support these, turning Snap into an actual programming language.
- Add some more types!
- We haven't even mentioned how to add a JIT to your interpreter. The RPython docs cover the basics. Your task will be to inform the RPython toolchain about a number of the fundamentals within your interpreter, such as where your main loop starts, and what the lifecycles of some of your variables and objects are within your interpreter.

CHAPTER 12

Indices and tables

- genindex
- search