yay Documentation

Release 3.1.2.dev0

John Carr

Sep 27, 2017

Contents

We wanted a confguration language that was human readable and human editable with variable expansion. We wanted something where we wouldn't keep hitting things that the language couldn't express.

Yay is both that language and a python module for parsing it.

Contents:

CHAPTER 1

Language Tour

Yay is a non-strict language that supports lazy evaluation. It is a sort of mutant child of YAML and Python, with some of the features of both.

There are some significant differences from YAML and this absolutely does not attempt to implement the more esoteric parts of YAML.

A particularly significant restriction is that keys may not contain whitespace. keys in a configuration language are expected to be simple bare terms. This also helpfully keeps the magic smoke firmly inside our parser.

It is important to understand that for any line of input it is imperative "pythonish" or declarative "yamlish". It actually works well and we find it very easy to read, for example:

```
a: b
if a == 'b':
    c: d
```

It is pretty clear that some of those lines are declarative and some are imperative. When in pythonish mode it works just as you would expect from python, when in yamlish mode it works as a declarative language for defining terms.

Mappings

A mapping is a set of key value pairs. They key is a string and the value can be any type supported by Yay. All Yay files will contain at least one mapping:

```
site-domain: www.yaybu.com
number-of-zopes: 12
in-production: true
```

You can nest them as well, as deep as you need to. Like in Python, the relationships between each item is based on the amount of indentation:

```
interfaces:
    eth0:
```

```
        interfaces: 192.168.0.1
        dhcp: yes
```

List

You can create a list of things by creating an intended bulleted list:

```
packages:
    - python-yay
    - python-yaybu
    - python-libvirt
```

If you need to express an empty list you can also do:

```
packages: []
```

Variable Expansion

If you were to specify the same Yaybu recipe over and over again you would be able to pull out a lot of duplication. You can create templates with placeholders in and avoid that. Lets say you were deploying into a directory based on a customer project id:

```
projectcode: MyCustomer-145

resources:
    - Directory:
        name: /var/local/sites/{{projectcode}}

    - Checkout:
        name: /var/local/sites/{{projectcode}}/src
        repository: svn://mysvnserver/{{projectcode}}
```

If you variables are in mappings you can access them using `.` as seperator. You can also access specific items in lists with `[]`:

```
projects:
  - name: www.foo.com
    projectcode: Foo-1
    checkout:
        repository: http://github.com/isotoma/foo
        branch: master

resources:
    - Checkout:
        repository: /var/local/sites/{{projects[0].checkout.repository}}
```

Sometimes you might only want to optionally set variables in your configuration. Here we pickup `project.id` if its set, but fall back to `project.name`:

```
project:
    name: www.baz.com

example_key: {{project.id else project.name}}
```

Including Files

You can import a recipe using the yay extends feature. If you had a template `foo.yay`:

```
resources:
    - Directory:
          name: /var/local/sites/{{projectcode}}
    - Checkout:
          name: /var/local/sites/{{projectcode}}/src
          repository: svn://mysvnserver/{{projectcode}}
```

You can reuse this recipe in `bar.yay` like so:

```
include "foo.yay"

include foo.bar.includes

projectcode: MyCustomer-145
```

Search paths

You can add a directory to the search path:

```
search "/var/yay/includes"

search foo.bar.searchpath
```

Configuration

::

configure openers:

foo: bar baz: quux

configure basicauth: zip: zop

Ephemeral keys

These will not appear in the output:

```
for a in b
    set c = d.foo.bar.baz
    set d = dsds.sdsd.sewewe
    set e = as.ew.qw
    foo: c
```

Extending Lists

If you were to specify resources twice in the same file, or indeed across multiple files, the most recently specified one would win:

```
resources:
    - foo
    - bar

resources:
    - baz
```

If you were to do this, resources would only contain baz. Yay has a function to allow appending to predefined lists: append:

```
resources:
    - foo
    - bar

extend resources:
    - baz
```

Conditions

```
foo:
    if averylongvariablename == anotherverylongvariablename and \
        yetanothervariable == d and e == f:

      bar:
        quux:
            foo:
                bar: baz

    elif blah == something:
        moo: mah

    else:
      - baz
```

For Loops

You might want to have a list of project codes and then define multiple resources for each item in that list. You would do something like this:

```
projectcodes:
    MyCustomer-100
    MyCustomer-72

extend resources:

    for p in projectcodes:
        - Directory:
```

```
            name: /var/local/sites/{{p}}

        for q in p.qcodes:
            - Checkout:
                name: /var/local/sites/{{p}}/src
                repository: svn://mysvnserver/{{q}}
```

You can also have conditions:

```
fruit:
    - name: apple
      price: 5
    - name: lime
      price: 10

cheap:
    for f in fruit if f.price < 10:
        - {{f}}
```

You might need to loop over a list within a list:

```
staff:
  - name: Joe
    devices:
      - macbook
      - iphone

  - name: John
    devices:
      - air
      - iphone

stuff:
    for s in staff:
        for d in s.devices:
            {{d}}
```

This will produce a single list that is equivalent to:

```
stuff:
  - macbook
  - iphone
  - air
  - iphone
```

You can use a for against a mapping too - you will iterate over its keys. A for over a mapping with a condition might look like this:

```
fruit:
  # recognised as decimal integers since they look a bit like them
  apple: 5
  lime: 10
  strawberry: 1

cheap:
    for f in fruit:
       if fruit[f] < 10:
         {{f}}
```

That would return a list with apple and strawberry in it. The list will be sorted alphabetically: mappings are generally unordered but we want the iteration order to be stable.

Select

The select statement is a way to have conditions in your configuration.

Lets say `host.distro` contains your Ubuntu version and you want to install difference packages based on the distro. You could do something like:

```
packages:
    select distro:
        karmic:
            - python-setuptools
        lucid:
            - python-distribute
            - python-zc.buildout
```

Function calls

Any sandboxed python function can be called where an expression would exist in a yay statement:

```
set foo = sum(a)
for x in range(foo):
    - x
```

Class bindings

Classes can be constructed on-the-fly:

```
parts:
    web:
        new Compute:
            foo: bar
            % for x in range(4)
                baz: x
```

Classes may have special side-effects, or provide additional data, at runtime.

Each name for a class will be looked up in a registry for a concrete implementation that is implemented in python.

Macros

Macros provided parameterised blocks that can be reused, rather like a function.

you can define a macro with:

```
macro mymacro:
    foo: bar
    baz: {{thing}}
```

You can then call it later:

```
foo:
    for q in x:
        call mymacro:
            thing: {{q}}
```

Prototypes

Prototypes contain a default mapping which you can then override. You can think of a prototype as a class that you can then extend.

In their final form, they behave exactly like mappings:

```
prototype DjangoSite:
    set self = here

    name: www.example.com

    sitedir: /var/local/sites/{{ self.name }}
    rundir: /var/run/{{ self.name }}
    tmpdir: /var/tmp/{{ self.name }}

    resources:
        - Directory:
            name: {{ self.tmpdir }}

        - Checkout:
            name: {{ self.sitedir}}
            source: git://github.com/

some_key:
    new DjangoSite:
        name: www.mysite.com
```

Here

Here is a reserved word that expands to the nearest parent node that is a mapping.

You can use it to refer to siblings:

```
some_data:
    sitename: www.example.com
    sitedir: /var/www/{{ here.sitename }}
```

You can use it with `set` to refer to specific points of the graph:

```
some_data:
    set self = here

   nested:
       something: goodbye
       mapping: {{ self.something }}          # Should be 'hello'
       other_mapping: {{ here.something }}    # Should be 'goodbye'
```

```
something: hello
```

CHAPTER 2

Encrypted Settings

If your yay config has a .gpg extension, Yay will attempt to decrypt it with GPG.

You would generate the .gpg version of a yay file using the GPG command line tools.

    $ gpg -e s00persekrit.yay

This will encrypt with public key / private key encryption. It will prompt you for recipients. These are the GPG keys that can decrypt your secrets.

Your encrypted `s00persekrit.yay.gpg` might contain:

```
sekrit:
  username: admin
  password: password55
```

You can reference this as though it is an ordinary .yay file:

```
.includes:
  - s00persekrit.yay.gpg

myothervariable: The password is {{sekrit.password}}.
```

CHAPTER 3

Using yay in your code

Basic use

There are 2 ways to work with the Yay configuration language. The primary method is one that embraces the lazy nature of the configuration language but is still pythonic.

Consider the following configuration:

```
network:
    proxy:
        type: socks
        host: 127.0.0.1
        port: 8000

    allowed:
      - 127.0.0.1
```

You can setup yay to load such a file like so:

```
import yay, os
config = yay.Config(searchpath=[os.getcwd()])
config.load_uri("example.yay")
```

You can access keys as attributes:

```
proxy_type = config.network.proxy.type
```

You can also treat it like a dictionary:

```
proxy_type = config['network']['proxy']['type']
```

You can also iterate over them:

```
for allowed_ip in config.network.allowed:
    do_stuff(allowed_ip)
```

It will throw a NoMatching exception if the list isn't defined. If a list is optional you can catch the exception:

```
try:
    for allowed_ip in config.network.allowed:
        firewall.add(allowed_ip)
except errors.NoMatching:
    firewall.deny_all()
```

All node access is lazy. When you look up a key the object that is returned is a proxy. A promise that when the graph is result the key that you have requested will be returned. You should use this to do strict type checking coercion:

```
config.network.proxy.as_int()
```

This would raise a `errors.TypeError` because `network.proxy` is a dictionary not an integer.

You can use the more python `int()`, `float()` and `str()` operators:

```
print "address:", str(config.network.proxy.host)
print "port:", int(config.network.proxy.port)
```

However one advantage of the `node.as_` form is that you can express defaults in your python code:

```
print "address:", config.network.proxy.host.as_string(default='localhost')
print "port:", config.network.proxy.port.as_int(default=8000)
```

If your code detects an error in the configuration it can use the anchor property of the node to raise a useful error message:

```
proxy_type = config.network.proxy.type
if not str(proxy_type) in ("socks", "http"):
    print "Incorrect proxy type!"
    print "file:", proxy_type.anchor.source
    print "line:", proxy_type.anchor.lineno
    print "col:", proxy_type.anchor.col
```

Finally, if you just want to examine your configuration via python simple types you can use the `yay.load_uri` and `yay.load` API's:

```
config = yay.load_uri("example.yay")
assert isinstance(config, dict)
```

The disadvantage of this approach is you lose access to line/column metadata for error reporting.

Writing a 'graph program'

You can factor your application as a series of graph objects that are plumbed together using the Yay language.

For example, in example_module/weather.py:

```
from yay.ast import PythonClass
from weather import Api

class Weather(PythonClass):

    def apply(self):
        api = Api(
            token = str(self.params.token),
```

```
            secret = str(self.params.secret),
            )

        for location in self.params.locations:
            self.metadata[str(location)] = api.lookup(str(location))
```

This can then be consumed from your configuration like so:

```
weather:
    create "example_module.weather:Weather":
        token: mytoken
        secret: mysecret
        locations:
            - york

# Deploy a funny MOTD when its cold
if weather.york.temperature < 0:
    extend resources:
        - File:
            name: /etc/motd
            static: motd/motd.cold
else:
    extend resources:
        - File:
            name: /etc/motd
            static: motd/motd
```

With this API your classes only need to do the following:

- Descend from `yay.ast.PythonClass`
- Implement the `apply` method. This is where you can look things up and change the external environment - for example, you might use `libcloud` to start a cloud compute node.
- To access settings passed to you (effictively as constructors), use `self.params`. This has the same interface as described in the previous section.
- To expose new metadata you have generated in your apply method, set it on the metadata dictionary. These will be 'bound' as Yay objects as they are accessed so you can just put basic python types in the dictionary and it will do the rest.

CHAPTER 4

Debugging

Viewing AST as a digraph

You can dump a yay graph using the `yay-dump` console script:

```
yay dot foo.yay > foo.dot
```

On a Mac, if you have GraphViz installed you can open this in a viewer with the open command:

```
open foo.dot
```

CHAPTER 5

Introduction

The YAY grammar is based on Python, with additional productions to provide for the data language stanzas that surround and enclose the pythonic terms. The data language resembles YAML, but it does not implement much of the complexity of YAML, which we consider unnecessary for this use case.

The lexer and parser are implemented using PLY.

CHAPTER 6

The Scanner

Input text is tokenised by a stateful lexer. It is a relatively traditional sort of scanner, except for the interpretation of leading whitespace. Leading whitespace is processed and consumed using a generator chain, to yield INDENT and DEDENT symbols as necessary. Other whitespace outside string literals is eliminated entirely from the output sequence.

The scanner has six states that drastically change how input text is interpreted.

INITIAL In this state the scanner is looking for keys in the YAML-like data language, or for reserved words that indicate the line is pythonic.

VALUE A key has been emitted and the scanner is now looking for a value.

LISTVALUE We are parsing a list of values, each introduced with a '-'

TEMPLATE We are within a template (i.e. {{ }}) and so should emit everything read as-is

COMMAND We are within a "command", i.e. a line of something pythonic

BLOCK We are within a multiline literal block

CHAPTER 7

Productions

The docstrings in parser.py provide the productions themselves, in the syntax required by PLY. Here we summarise the productions in a slightly more typical syntax, and without the NEWLINE, INDENT and DEDENT symbols that are vital for parsing but only get in the way when trying to understand the grammar.

Python

The parser implements virtually all of the Expressions defined for Python:

http://docs.python.org/2/reference/expressions.html

As well as many of the compound statements:

http://docs.python.org/2/reference/compound_stmts.html

Data Language

In addition to these are the data language itself:

Stanzas

The root of a document consists of zero or more stanzas. Each stanza is one of:

```
stanza ::= yaydict
         | yaylist
         | extend
         | directives
```

Extend

An extend looks like:

```
extend ::= "extend" key ":" scalar
         | "extend" key ":" stanzas
```

Scalars

A scalar is basically a VALUE read from the data language, or more than one value. It also handles literal empty dictionaries and merges multiline symbols from the scanner.

scalar ::= "{}"

"[]"
value
"{{" expression_List "}}"
multiline
scalar+

Dictionaries

These provide the mapping objects in the language:

```
yaydict ::= key ":" scalar
          | key ":" stanza
          | key ":"
          | yaydict+
```

Lists

These provide the sequence objects in the language:

```
yaylist ::= listitem+

listitem ::= "-" scalar
           | "-" key ":" scalar
           | "-" key ":" stanza+
           | "-" key ":" scalar yaydict
```

The last three productions above handle the complex case of a dictionary as a list item.

Directives

Python's compound statements are called "Directives" within YAY. In addition to the standard python compound statements, YAY introduces a number of it's own directives:

```
directive ::= include_directive
            | search_directive
            | new_directive
            | prototype_directive
            | for_directive
```

```
            | set_directive
            | if_directive
            | select_directive
            | macro_directive
            | call_directive
```

Include

The include directive includes another file at this point in the input text:

```
include_directive ::= "include" expression_list
```

Search

This adds components to the search path:

```
search_directive ::= "search" expression_list
```

Prototype

Prototype defines a structure to be reused later:

```
prototype_directive ::= "prototype" expression_list ":" stanza+
```

New

New uses a previously defined prototype:

```
new_directive ::= "new" expression_list ":" stanza+
                | "new" expression_list "as" identifier ":" stanza+
```

For

This works like a python for statement:

```
for_directive ::= "for" target_list "in" expression_list ":" stanza+
                | "for" target_list "in" or_test "if" expression ":" stanza+
```

Set

Set allows temporary local variables to be used, without them appearing in the output graph as data symbols.

set_directive ::= “set” target_list “=” expression_list

If

If resembles the python conditional:

```
if_directive ::= "if" expression_list ":" stanza+ ("else:" stanza+)?
               | "if" expression_list ":" stanza+ ("elif" expression_list ":"␣
↪stanza+)+ ("else:" stanza+)?
```

Select

Select implements the select statement familiar from many other languages, but not present in Python. This is not required in Python, but is a common requirement in Yay:

```
select_directive ::= "select" expression_list ":" (key ":" stanza+)+
```

Macro

Macros are similar to prototypes, in that they allow for simple reuse. See the language documentation to understand how to use them:

```
macro_directive ::= "macro" target_list ":" stanza+
```

Call

Call invokes previously defined macros:

```
call_directive ::= "call" target_list ":" stanza+
```

CHAPTER 8

Resolving

The language is parsed into a directed acyclic graph, with each node representing some data or an expression that must be evaluated in order to produce some data. This sections describes in detail the properties of members of this graph structure.

Relationships

Parenting

Every node should have one and only one parent. Detached nodes are not valid. It is the responsibility of the parent to adopt its child nodes and let them know who there parents are.

Parents will hold a strong ref to their children. Therfore the ref back to the parent can be a weakref.

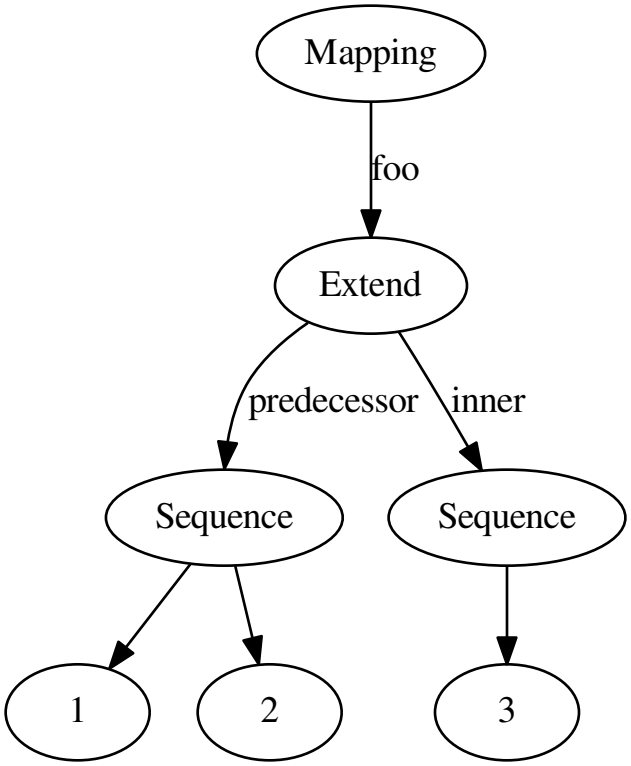
Predecessors

When a node is inserted into a Mapping it might supercede an existing key. The graph will keep track of this, which allows mappings to perform operations that involve previous versions of themselves. For example:

```
foo:
  - 1
  - 2

extend foo:
  - 3
```

This would parse to:



With this structure it is then easy to combine the original list with the 2nd list.

Root

Any node is able to find the root of the graph by following the parent edges to a leaf node.

Head & Successor

These are calculated references.

Consider the following:

```
foo:
    a: {{ here.b }}
foo:
    b: 1
```

Because of the inherent locality of graph scoping, the `here` variable is unable to find b.

Because the second `foo` block has a predecessor relationship with the first `foo`, we implicitly track the inverse `successor` relationship. `here` can therefore follow the `successor` chain to its tip (the `head`) and use that to lookup `b`.

Just-in-time visiting

In order to transform the graph into it's final form we need to perform a series of transformations on the graph.

The standard approach would be a series of transformations performed via the visitor pattern. Each visitor would know one transformation and they would be performed in order. This does approach to graph rewriting does not suit a graph that is as dynamic as ours:

1. We can't have a list of transformations to apply in order. For example, to simplify an `If` node one might need to have first simplified a `For` node. But also vice-versa.
2. You must resolve parts of the graph to even type check other parts of it. For example:

   ```
   foo: {{ 1 + 2 }}
   bar: {{ baz[foo] }}
   ```

   In order to even begin to speculate about some parts of the graph, other parts must be completely resolved. In order to build a standard dependency graph, some parts of the graph must be fully solved.

The conclusion here is that any graph transformation needs to happen just-in-time. Another way of thinking about this is that while a traditional visitor might visit the graph in the order it is parsed, we need to apply the transformations as a depth first exploration of its dependencies.

(FIXME: There are some really icky and hard to qualify things here, really hard to qualify, come back and fix this!)

The main problem with even a depth first visitor is one of context. It becomes harder to know what is appropriate to resolve and how far.

There are essentially 3 target states:

- Fully resolved - a python simple type like `str`, `list` or `dict`
- Folded - a graph safe simple type like `Boxed`, `Sequence` or `Mapping`
- Traversible - a graph safe type that is solved enough to allow it to be traversed

These states are more fully explored in the following sections. For now it is enough to know that they are target states. Some graph nodes will be in a 'folded' state from day 1 (like `Boxed`) and some will be 'traversible' without any transformations (like `Mapping`). 'Fully resolved' objects will never exist in the graph.

Given these rules what does a visitor look like when it has to make some nodes folded to make them traversible and resolve others to make their dependants foldable?

The simplest solution is that you don't use a visitor at all. Actually for our situation, each node just needs to know how to simplify itself into the various target states and it needs to know what state its dependents need to be in in order to reach its target state.

For example, consider a node that sums 2 dependent graph members:

```
class Addition(object):
    def __init__(self, dependentA, dependentB):
        self.a = dependentA
        self.b = dependentB
    def traversible(self):
        self.error(NotTraversible())
    def folded(self):
        # We explicitly fold our dependencies and rely on exceptions to bail out when␣
↪something is unfoldable
        # This is covered in a later section
        a, b = self.a.fold(), self.b.fold()
        return Boxed(a.resolve() + b.resolve())
```

```
    def resolve(self):
        return self.a.resolve() + self.b.resolve()
```

Expanding (aka Traversible)

The power of Yay is its lazyness. In order to make the language sufficiently lazy the graph has to avoid resolving any data structues it can until the last moment.

A simple example is a nested mapping:

```
foo:
   bar: {{ some_other_section }}
   baz:
     qux: 1
     quix: 2
```

You shouldn't need to resolve `bar` (and hence the whole of `some_other_section`) to get to `baz`. That would rather limit the flexibility of lazy evaluation.

So mapping nodes can be traversed without needing to resolve the entire graph. We do this with the `get` function:

```
graph.get("foo").get("baz").get("quix").resolve() == 2
```

Things get a bit more complicated when command expressions are involved. Let's consider the `if` operation:

```
cond: hello
default: happy

if cond == "hello"
    default: really happy
    dont_resolve_me: {{ some.datastructure[0].somewhere.else }}
```

The parser will return an If node that has a predecessor. The If node needs to be traversal friendly. There is no need to resolve the `dont_resolve_me` variable when attempting to access `default`.

This is where the `expand` API comes in. In order to resolve `default` we need to resolve the guard expression. But there is no need to resolve the other child nodes of If.

In this case, calling expand() will return the predecessor mapping if the condition is false and the child mapping if it is true. In otherwords, the condition is resolved but the mapping that is guarded by the condition is not. We can then access `default` without triggering `dont_resolve_me`.

It is important that when a node is expanded the node that it returns is indeed expanded. To clarify, consider this example:

```
var: 1

% if 0:
    var: 2

% if 0:
    var: 3

foo: {{ var }}
```

If this was parsed and you attempted to expand `foo` we'd expect it to return a `Boxed(1)`

When the first `If` node is expanded it will realise that the condition is false and attempt to return its predecessor. However it's predecessor is a `If` node as well. So if when a node is expanded it returns another existing node it should take care to call `expand` upon it. In this case, the 2nd `If` will expand to a `Mapping` and when a `Mapping` is expanded it will just return itself. This is the correct behaviour.

Folding

Of course there are some nodes that cannot be simplified. A completely pure graph can be entirely solved to a single value. However (as discussed later in "Native Classes") not all graph members are pure. An extra stage is required to fully support these non-pure elements. We call this the folding step.

When the graph is folded we are essentially doing a traditional constant folding step that a compiler might do to try and generate better code. The graph is resolved to "simple types" like:

- Boxed
- Mapping
- Sequence

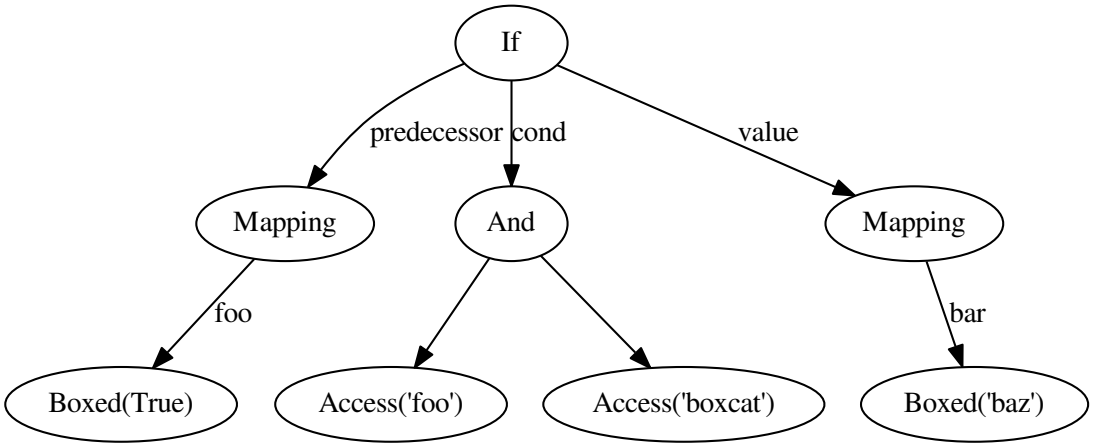
I.e. the goal is to remove any of the 'command mode' structures like `If` and `For`. The results are still in graph form - we haven't simplified them to python simple types.

However, non-pure graph members cannot be folded as we cannot know their value without causing side effects. Let's consider a variable `boxcat` that will be `True` or `False`. Our input is this:

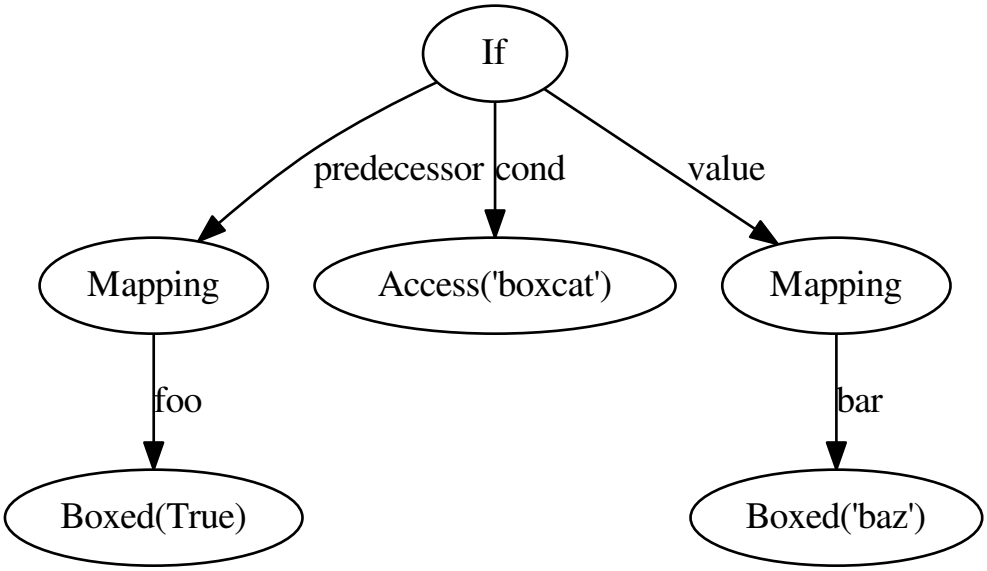
```
foo: True

% if foo and boxcat:
    bar: baz
```

The initial parsed form is:



The folded form is:



The first `Access` (to `foo`) has been simplified away, as has the `And` expression. The `If` node is still present because it depends on an unknown external value - `boxcat`. This graph is now as simple as it can be without suffering any side effects.

The implementation might look something like this:

```
class And(object):
    def folded(self):
        uleft, uright = True, True
        try:
            left = self.left.folded()
        except CantFold:
            left = self.left
            uleft = True
        try:
            right = self.right.folded()
        except CantFold:
            right = self.right
            uright = True

        if uright and uleft:
            raise CantFold

        elif uright and not uleft:
            if left.resolve():
                raise CantFold(right)
            else:
                return Boxed(False)

        elif uleft and not uright:
            if right.resolve():
                raise CantFold(left)
```

```
            else:
                return Boxed(False)

        else:
            return Boxed(left.resolve() and right.resolve())
```

Gnarly! But this is just an encapsulation of some really simple rules:

- If neither side of the `And` is a constant then we can't fold
- If both sides are then we can fold and return `True` or `False` via a `Boxed`
- Otherwise we can fold and resolve the constant side of the expression - If it is False then we can short ciruit the dependency on the external value and return `Boxed(False)` - If it is True then we can't fold, but we can simplify and remove both the `And` and the constant side of the expression

Variable expansion

Expressions can reference variables. These might be keys in the global document or they might be temporary variables in the local scope. An example of this might be:

```
somevar: 123

foo:
    % let temp1 = 123
    bar: {{ somevar }} {{ temp1 }}
```

In order to resolve `bar` the graph needs to be able to resolve `temp1` and `somevar`.

When a variable is referenced from an expression it is not immediately 'bound'. This is not the point at which we traverse the graph and find these variables. Instead we place an `Access` node in the graph.

Primarily an `Access` node needs to know the key or index to traverse to. This is an expression that will be resolved when any attempt to expand the node is actioned. This expression could be as simple as a literal, or as complicated as something like this:

```
{{ foo.bar[1].baz[someothervar[0].bar] else foo.bar[0] }}
```

When no additional parameters are passed to an Access node it will look up the key in the current scope (see the Context section).

However you can specify an expression on which to act. This is useful because you can chain several `Access` nodes together. For the example above, the expression `{{foo.bar}}` would be parsed to:

```
Access(Access(None, "foo"), "bar")
```

Context

The language has some variables that are scoped. For example:

```
i: 5

foo:
  % for i in baz
      - {{ i }}
```

```
```

`i` has different values depending on whether you are inside the for loop or not.

In early versions of yay context was handled by passing around a context object. Anytime a node contributed to the context it would push to this context object. This was problematic:

```
i: 5
b: {{i+1}}

foo:
  % for i in baz
      - i: {{ i }}
        b: {{ b }}
```

Is `b` always `6`, or does its value change with the for loop? The correct behaviour is that it is always 6 but a context object approach did not allow this.

Another disadvantage of this approach is that a node doesn't resolve to one state - it resolves to many states as it could be passed many different contexts. This makes memoization uglier and it caused suspicion that variables might change as the graph was resolved - this is not supposed to be possible.

The current approach is to treat context as a member of the graph. When an object wants to look up a name and consider scope it asks its parent for the nearest context node. This just traverses its parents until it reaches a context node or reaches the root of the graph. If a context node cannot answer it's query then traversal continues. When the root of the graph is reached if no match has been found the `get` method is called on the root. This may raise an exception if there is no such node.

If

An `If` node will resolve a guard condition and if it is `True` then the contents of the node apply, otherwise attempts to use it will be proxied to its `predecessor`.

In order to be traversed the guard condition has to be resolved.

When folded the node will try to fold the guard condition and if it cannot be folded then the if statement itself will not be factored away. However the guard condition may still be partially simplified, as may the contents of the child node.

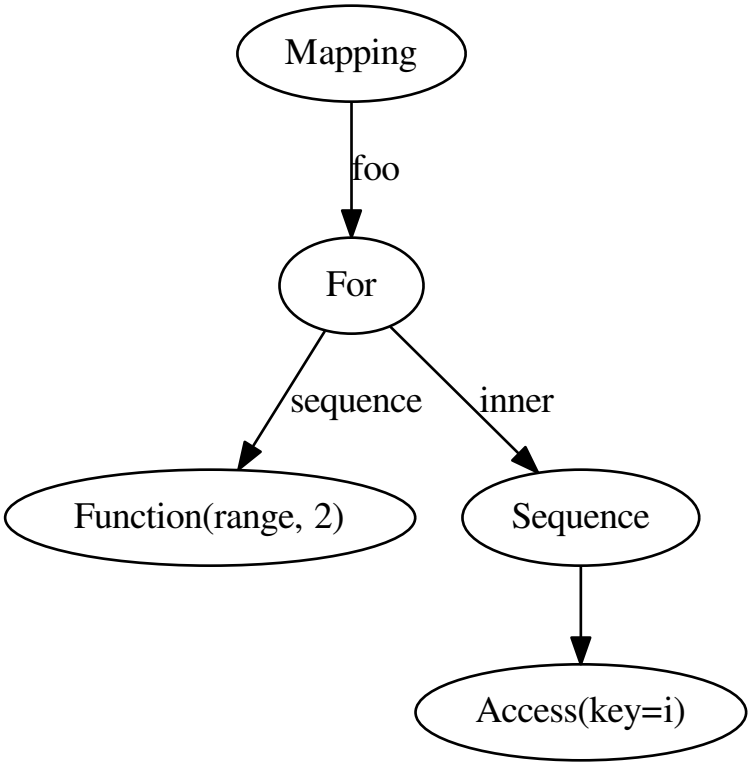
For

The expansion of a for loop requires its children to be cloned and parented to a context node for each iteration of the loop. For example:

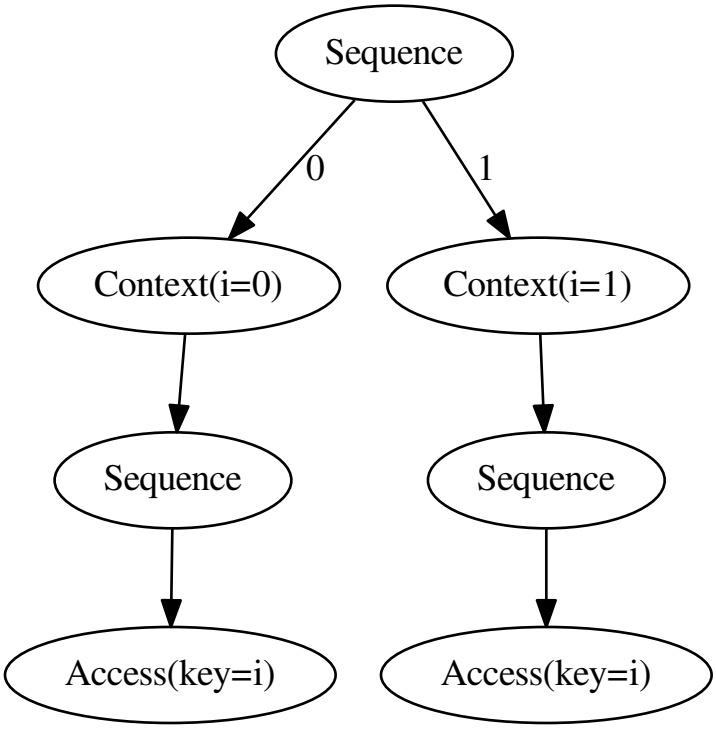
```
baz:
  - 1
  - 2

foo:
  % for i in baz
      - {{ i }}
```

This would parse to:



This might expand to:



Native Classes

You can bind custom code to the yay graph that interfaces with code outside the graph. Code wrapped for consumption by our non-strict graph is called an 'Actor'. (FIXME: This is subject to change, but Actor is better than further complicating terms like 'Node').

By allowing an engineer to bind their side-effect causing code directly to the graph we gain quite a few powerful features:

- Implicit dependency graph of relationships between actors
- Implicit ability to parallelize actor side effects (e.g. load balancer with 20 backends - we can deploy those backends in parallel)

However there are consequences:

- It is impossible to completely validate the graph ahead of time (doing so would require us to actually cause our side effects)

Actor nodes must follow certain rules so that we can maximise the safety of any operations.

It is clear that in order to avoid activating the native code too soon they need to be the laziest kind of graph member. This is the main reason for the folding step.

Incredibly lazy importing

One feature of yay1 was that imports were immediate but could consume lazy variables. For example:

```
.include: cookbook/entrypoints/${foo}.yay
```

The consequence of this is that `foo` was consumed mid-way through loading the config. But it could be overloaded later in the config. So while `foo` might have been `apache` when the include was processed, it might be `gunicorn` by the time the config is fully parsed. The crude work around was that any variable used to satisfy an include would be 'locked'. Any further attempts to modify that variable should be met with horror.

In yay3 we defer parsing until required.

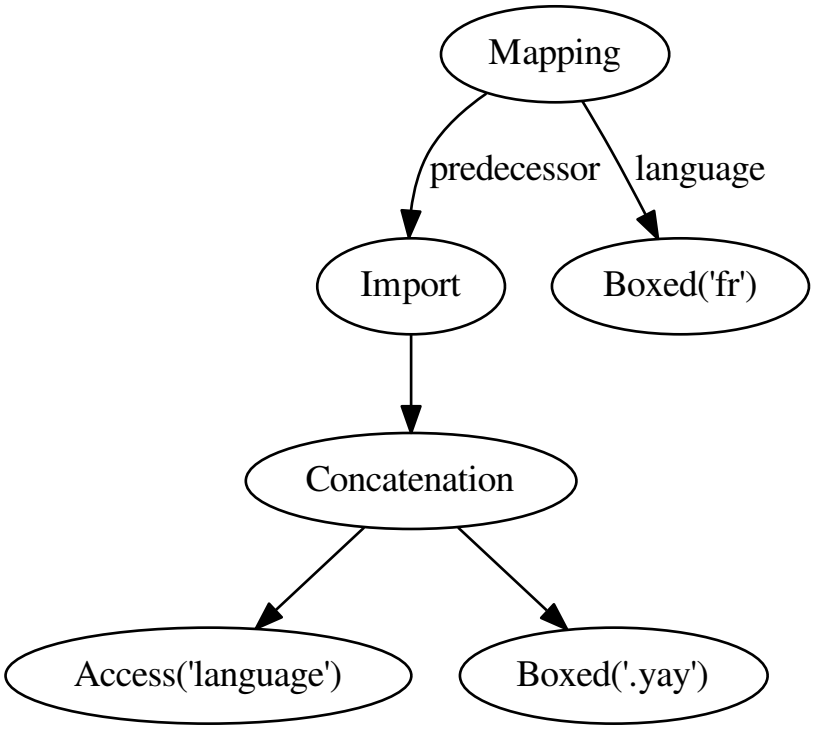
Imagine 2 simple yay documents. The first is `fr.yay`:

```
hello_world: Bonjour!
```

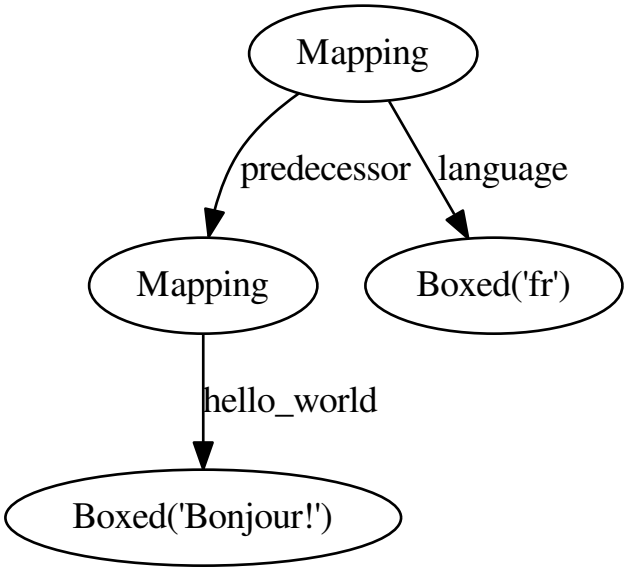
And the second is `main.yay`:

```
% import {{ language }}.yay
language: fr
```

An initial parsing of this might be:



After constant folding this would expand to:



Because we have removed the need to process the import immediately we no longer have complex document locking requirements.

Early Error Detection

When not using the class feature of yay then early error detection is not useful. Detecting all errors will cause the graph to be resolved any way, so might as well be done JIT.

However the current approach for 'nodes with side effects' means that you might not have even finished syntax checking before you have started mutating an external system. In this case, any additional checking you can do is worth it.

The topics discussed in this section are currently in the idea stage. Navigating the graph without triggering premature expansion is tricky.

Type fixing

One type of analysis that we can perform on the graph is to look at the predecessors of each node and make sure that the types of fields don't change. Once a number, always a number.

For these purposes the only types that matter are:

```
* Number
* String
* List
* Dict
```

Some type inference is possible:

- We know that a foreach will resolve to a list.
- If a variable resolves to a constant, then we can get its type - we can do this without causing resolves in some cases.

However there are problems.

Consider a case like this:

```
foo: bar
qux: quux

if somexpr:
    foo: []

qux: fozzle
```

The only way to be certain if the final config is correct is to resolve `somexpr`. This could in the worst case actually cause a side effect.

Another possibility is to have speculative type inference: The if knows it might return a list for `foo` or it might have to defer to its predecessor. However actually implementing that might be difficult...

Schemas

Part of the problem with external sources of information is we don't know what outputs they have. If we require nodes to declare their inputs and outputs then we can do additional checking. This is actually what we do with `Resources` in yaybu atm - there is a schema system in yaybu.

Short term problems to solve

- Need to consider `.get()` - in particular how it interacts with `traversible`. My worry is things that need to be resolved to traverse an `If` node might be dynamic and side effect causing. The rules there need qualifying here, I think.

Future Work

Parallelization

The goal here would be to maximise the amount of work that is done in parallel. One way to achieve that is to make it OK for a resolve to end prematurely with a `ResultNotReady` exception. When that happens the exception would generally be bubbled up to the root node. However containers could try and resolve their other children at this time. A mapping could resolve its other keys. A sequence could resolve siblings of the node that isn't ready. The result of this would be that 'Actor' nodes could perform side effects in parallel.

This probably shouldn't be tied to twisted - we don't want to complicate supporting gevent or blocking use cases.

Online graphs

The key problem with a live graph is that data flows are push rather than pull based, and this inverts some of the approaches we'd normally take.

In particular, our approach to `{{ binding.a.variable }}` needs rethinking. At the very least we need to have bound variables be notified which part of the graph is trying to `.get()` them. That way, if and when they change we can notify any neighbouring regions to readjust.

Allowing the graph to 'settle' when dealing with events its also an interesting challenge. Several event sources saturating the graph with information will fail in much the way a human mind would - too much information, too many plates spinning. One event could impact multiple 'Actor' nodes. Those actor nodes might depend on each other, and some of the cloud services have sllllloooow API's. So an event/second could quickly be too much.

If an online graph is what is most desirable that it is worth considering ways to remove the k/v underpinnings. Maybe the k/v rooted approach is actually just scaffolding to wire a big pipeline. What if our airship didn't need that scaffolding when we had done the initial parse. The notion of outputting a YAML like document of this monster could actually limit how we build some of the interfaces...

Event to scalar

Typical simple graphs are run once and then discarded. However with a robust graph API in place we can use yay as a live decision system. Consider an external data source that subscribes to events from ZeroMQ:

```
metrics:
    web_load:
        % ZeroMQ
            connect: mq.example.com
            subscribe: {{ cluster.name }}_load_web

loadbalancer:
    % LoadBalancer
        listen: http
        members:
            % for i in range(load_to_boxen_needed(metrics.web_load))
                % Compute
                    name: web{{i}}
                    cookbook: entrypoints/web.yay
```

The relationship between a metric and the number of compute nodes isn't interesting so i've just black-boxed it with a function. This graph is interesting because `metrics.web_load` is sourced from ZeroMQ. It can and will change over time and we can potentially have a graph that responds to external changes...

Events to lists

Imagine a database that holds information about services in a cluster. If we represented it with YAML we might see:

services:

- name: 1.example.com type: node.js
- name: 2.example.com type: php

Imagine this data can optionally be fed from ZeroMQ or a Websocket so it is always up to date. We can actually feed those events in to the initial list state to keep it up to date, those nodes then in turn notify other nodes.

In other words, as soon as you add a new site to your configuration management system several nodes would be notified that their configuration was out of date and that they needed to redeploy. But because of the magic of the graph, it would be only the nodes that needed to be updated!

Gotchas / Headscratching

Think about:

- It's easy enough to have a graph node that listens for changes, but what does that actually do? The graph API is pull based. *Right now* we can't push a new value down the chain.
- I think changes that are detected notify the root node.
- That node will then resolve itself.
- The number of compute nodes then may or may not be changed based up the external event.

Terminology

The following terms are used in the section, at times without proper thought as to terminology clashes with other yay modules and often with a complete lack of regard for any traditional use of the term.

Node An element or member of the solver graph

Predecessor An edge (or arc) to an ancestor. Consider key `foo`. Regardless of how many times you assign a value to it, all of those values are still accessible from the graph by walking the `predecessor` edges. Thus a node that has a `predecessor` of `None` is the oldest version of a key.

Parent All but the root node of the graph are contained within a `parent` node.

Actor Node A member of the graph that causes external code to be executed - potentially causing side effects.

Expression Node A member of the graph that is resolved by performing an expression against other members of the graph. For example, `1 + 1` or `1 + foo`.

Data Node Mapping, sequence or literal

Command Node A statement block such as `if` or `for`

CHAPTER 9

Contributing

To contribute to Yay send us pull requests on GitHub!

If you have any questions we are in `#yaybu` on `irc.oftc.net`.