Yaml.rb -- Yaml for Ruby

---

by why the lucky stiff

[Version 0.49]

Table of Contents

1. Preface

What is YAML? From the specification:

> *YAML(tm) (rhymes with 'camel') is a straightforward machine parsable data serialization format designed for human readability and interaction with scripting languages such as Perl and Python. YAML is optimized for data serialization, formatted dumping, configuration files, log files, Internet messaging and filtering. This specification describes the YAML information model and serialization format. Together with the Unicode standard for characters, it provides all the information necessary to understand YAML Version 1.0 and construct computer programs to process it.*

For Ruby developers, YAML is a natural fit for object serialization and general data storage. Really, it's quite fantastic. Spreads right on your Rubyware like butter on bread!

The possible uses for YAML are innumerable. Configuration files, custom internet protocols, documentation, the list goes on and on. Also, with YAML readers popping up for other languages (see YAML.pm and others), you can pass data easily to colleagues in distant lands, swamped in their archaic languages.

YAML is a beacon of light, reaching out to them all. ;)

If I can (quickly, of course) in the Pickaxe book, an all-time favorite coding book, Dave Thomas and Andy Hunt say:

> *When we discovered Ruby, we realized that we'd found what we'd been looking for. More than any other language with which we have worked, Ruby stays out of your way. You can concentrate on solving the problem at hand, instead of struggling with compiler and language issues. That's how it can help you become a better programmer: by giving you the chance to spend your time creating solutions for your users, not for the compiler.*

HeiL! So true. Ruby's elegance, its readability, its common sense! Such it is with YAML. YAML is completely readable, in fact much of its syntax parallels Ruby's own data structure syntax!

Another one from the Pickaxe:

> *Ruby is easy to learn. Everyday tasks are simple to code and once you've done them, they are easy to maintain and grow. Apparently difficult things often turn out not to have been difficult after all. Ruby follows the Principle of Least Surprise--things work the way you would expect them to, with very few special cases or exceptions. And that really does make a difference when you're programming.*

A quick look at YAML and you can see your data structure immediately. If I compare it to SOAP or XML-RPC, the difference is immense. With XML-RPC, you can see the data structures, but its terribly verbose. More time is spent describing the structure than anything else. Again, the Principle of Least Surprise is wholly present in YAML.

2. Getting Started

2.1. Introduction

This Introduction will cruise you through the features of YAML.rb, exploring both YAML examples and accompanying Ruby code.

First, we'll go into how to convert data structures into YAML. Then, we'll talk about how to get data back out of YAML. We'll talk about some parts of YAML.rb that are customized to Ruby, then finish up with some information about how to get more help and some contact information.

2.2. Collections in YAML

When you break it all down, YAML is comprised of collections. Collections come in the form of sequences (Arrays) and mappings (Hashes).

Sequences

Here is an example of a simple sequence:

```
- dogs
- cats
- badgers
```

Ex. 1: Simple sequence in YAML

The above sequence is a set of three strings. When you load this YAML document into Ruby, you should receive the following Array:

```
[ 'dogs', 'cats', 'badgers' ]
```

Ex. 2: Simple sequence in Ruby

Every Object in Ruby should have a to_yaml method. This method can be called to dump an object into YAML. For example, we can turn the above Array into YAML like so:

```
require 'yaml'
puts( [ 'dogs', 'cats', 'badgers' ].to_yaml )
# prints:
#   - dogs
#   - cats
#   - badgers
```

Ex. 3: Using to_yaml to output a sequence

YAML sequences can contain any type of YAML data, including other sequences and mappings. To nest a sequence within another sequence, simply begin a new level of indentation:

```
-
  - pineapple
  - coconut
-
  - umbrella
  - raincoat
```

Ex. 4: Nested sequence in YAML

Mappings

Now, let's look at a simple mapping, the other type of collection:

```
dog: canine
cat: feline
badger: malign
```

Ex. 5: Simple mapping in YAML

The above mapping is a set of three key/value pairs. In Ruby, this mapping would become a Hash:

```
{ 'dog' => 'canine',
  'cat' => 'feline',
  'badger' => 'malign' }
```

Ex. 6: Simple mapping in Ruby

The Hash also has a to_yaml method, which can be used to export a Hash object into YAML:

```
require 'yaml'
puts( { 'dog' => 'canine',
        'cat' => 'feline',
        'badger' => 'malign' }.to_yaml )
# prints:
#   dog: canine
#   cat: feline
#   badger: malign
```

Ex. 7: Using to_yaml to output a mapping

Like sequences, mapping values can be any type of object. Mappings can contain nested sequences and mappings.

```
Joey:
  age: 22
  sex: M
Laura:
  age: 24
  sex: F
```

Ex. 8: Nested mapping

Sequences and mappings may be quite simple, perhaps too simple for much of what you do as a programmer. The advantage to these collections is that they are supported by every YAML implementation. Which means that this data will be available in Python, Perl, Java or any other language which has a YAML library available.

Please refer to the YAML Cookbook [http://yaml4r.sourceforge.net/cookbook/] for more information on collections, including the abbreviated syntax for inline collections.

2.3. Basic Types in YAML

Strings, integers, floats, timestamps. These are the types of data that our data structures are fundamentally constructed from. YAML supports many of the basic types which are included in Ruby's standard library.

Single-line types

The Null, Boolean, Integer, Float, Time, and Date types each fit on a single line and map directly to their Ruby counterparts. Here is a sequence containing each of these types, respectively:

```
- ~
- true
- 10
- 10.2
- 2002-08-15T17:18:23.18-06:00
- 2002-08-15
```

Ex. 9: Basic types in YAML

In the above example, the tilde '~' character represents a NilClass object in Ruby. Here's a basic mapping showing some valid YAML elements and their corresponding Ruby classes:

```
~: NilClass
+: TrueClass
true: TrueClass
True: TrueClass
-: FalseClass
false: FalseClass
False: FalseClass
0: Integer
1: Integer
100: Integer
1,000: Integer
0.0: Float
1.0: Float
100.001: Float
1,000.001: Float
1.00009e+3: Float
2002-08-15T17:18:23.18-06:00: Time
```

```
    2002-08-15 17:18:23.18 -06:00: Time
    1976-07-31: Date
```

Ex. 10: Basic types and their Ruby classes

Basic types have their own to_yaml method (like any other object in Ruby), which can be used to generate YAML:

```
    puts( nil.to_yaml )
    # prints:
    #   --- ~
    puts( true.to_yaml )
    # prints:
    #   --- true
    puts( false.to_yaml )
    # prints:
    #   --- (false)
    puts( 10.to_yaml )
    # prints:
    #   --- 10
    puts( 20.45.to_yaml )
    # prints:
    #   --- 20.45
    puts( Time.now.to_yaml )
    # prints:
    #   --- 2002-08-15T17:29:01.79-06:00
    puts( Date.new( 1976, 07, 31 ).to_yaml )
    # prints:
    #   --- 1976-07-31
```

Ex. 11: Using to_yaml with basic types

The triple dash '---' in the above outputs is the YAML separator. Documents which contain a single collection (such as the Collections examples on the last page) don't require a separator. The above examples are not collections, so they require a separator.

Basic types which begin with an alphanumeric character are Strings if they don't fall into one of the above categories.

Multi-line types

Strings which span several lines can be represented in YAML as blocks. Blocks begin with either a literal '|' character or a folded '>' character. The block is then dumped into a new level of indentation:

```
    - literal: |
        A literal block keeps all
        new lines when it is brought
        into Ruby.
    - folded: >
        A folded block will get rid
        of its newlines, trading them
        for spaces when it is brought
        into Ruby.
```

Ex. 12: Literal and folded blocks

The String class has a to_yaml method which will determine how to best flow your text in the YAML document. By default, it will attempt to fold your text. If your text has indented portions, it will leave the text as-is and present it as a literal block:

```
txt = "Just got my (dead-tree, printed-on-paper, I don't know if there's a web " +
      "version) copy of Linux Magazine for September, 2002.  There's an article by " +
      "Dave Thomas about building networked applications in Ruby.\n\nProps to Dave!"
puts txt.to_yaml
# prints:
#   --- >-
#   Just got my (dead-tree, printed-on-paper, I don't know if there's a web version)
#   copy of Linux Magazine for September, 2002.  There's an article by Dave Thomas
#   about building networked applications in Ruby.
#
#
#   Props to Dave!
```

Ex. 13: Case where to_yaml generates a folded block

```
txt =<<EOF
ZenWeb 2.11.0 has been released!
I don't think I've remembered to announce any releases in a while, so
this one is a tad different. Among the newest changes are:
  + relative url renderer
  + massively improved demo/docs
:)
EOF
puts txt.to_yaml
# prints:
#   --- |
#
#   ZenWeb 2.11.0 has been released!
#
#   I don't think I've remembered to announce any releases in a while, so
#   this one is a tad different. Among the newest changes are:
#
#     + relative url renderer
#     + massively improved demo/docs
#
#   :)
#
```

Ex. 14: Case where to_yaml generates a literal block

Blocks are great because you can use all of the indicator characters freely without needing to escape them. This is actually an incredible advantage to YAML, as YAML documents can contain other YAML documents without needing to encode them! Think of what trouble you have to go through to include XML inside of XML!

Please see the YAML Cookbook for more information on the various string and block types.

2.4. Objects in YAML

YAML.rb has means for exporting custom objects to YAML. We've just gone over most of the types that YAML comes with, so we'll start off with some Ruby-specific types supported by YAML.rb. Then we'll cover how YAML.rb represents your custom classes.

Symbols

Ruby Symbols are stored as strings with a '!ruby/symbol' (or '!ruby/sym') type applied:

```
simple symbol: !ruby/symbol Simple
shortcut syntax: !ruby/sym Simple
symbols in maps:
  !ruby/sym MapKey: !ruby/sym MapValue
```

Ex. 15: Symbols in YAML

Naturally, Symbols have their own to_yaml method:

```
puts :Simple.to_yaml
# prints:
#   --- !ruby/sym Simple
```

Ex. 16: Using to_yaml with Symbols

Ranges

Ranges are store as strings with a '!ruby/range' type. The string is syntactically identical to the Range syntax in your Ruby code:

```
normal range: !ruby/range 10..20
exclusive range: !ruby/range 11...20
negative range: !ruby/range -1..-5
? !ruby/range 0..40
: range as a map key
```

Ex. 17: Ranges in YAML

You may use any YAML basic type in your Range, including Integers, Floats, Dates, and Timestamps.

```
floats: !ruby/range 10.2..21.5
dates: !ruby/range 2001-03-02..2001-09-11
```

```
timestamps: !ruby/range 2001-09-03T05:16:23Z..2001-09-04T07:08:13Z
```

Ex. 18: Other basic types in Ranges

Who knows how useful having other types in your Ranges really is, but it does illustrate the usefulness of YAML.rb's check_implicit, which allows built-in data types to be reparsed in custom types.

Regexps

Not all languages have built-in regular expression support. But for Ruby, regular expressions are a core object, an essential object!

YAML.rb uses YAML's typing mechanism to support the Regexp class. Regexps are represented as a string using the '!ruby/regexp' type:

```
starts with a b: !ruby/regexp '/^b/'
ends with a z: !ruby/regexp '/z$/'
search for a C or c: !ruby/regexp '/c/i'
```

Ex. 19: Regexps in YAML

Regexps also have a to_yaml method, which will type and quote the Regexp for you:

```
puts( /^b/.to_yaml )
# prints:
#   --- !ruby/regexp "/^b/"
```

Ex. 20: Using to_yaml with Regexps

Objects

As mentioned previously, every Ruby object has a to_yaml method which is introduced when YAML.rb is loaded. This method will generated a generic YAML representation of any Ruby object. The 'ruby/object' type will be used, along with the class name and a dump of the public members of the object.

As a simple example, let's suppose you have a Video class, which you are going to use to organize your Important-Enough-to-Write-a-Ruby-Script-for video collection. Here's your prototype class:

```
class Video
  attr_accessor :title, :year, :rating
  def initialize( t, y, r )
    @title = t
    @year = y
    @rating = r
  end
end
```

Ex. 21: Video class

You own two videos, so you create an Array of Videos to represent your video collection.

```
collection = [
  Video.new( "Ishtar", 1987, 8.8 ),
  Video.new( "Dr. Strangelove", 1964, 10.0 )
]
```

Ex. 22: Your Video collection

When you export the collection to YAML, you'll see each of the videos appear as a '!ruby/object:Video' type:

```
puts collection.to_yaml
# prints:
#   - !ruby/object:Video
#     title: Ishtar
#     year: 1987
#     rating: 8.8
#   - !ruby/object:Video
#     title: Dr. Strangelove
#     year: 1964
#     rating: 10.0
```

Ex. 23: Your Videos in YAML

You can actually code your own to_yaml method, as YAML.rb has a rather simple API for doing so. It's the same API used to build Yod, the software which generates this documentation.

2.5. The Options Hash

Whether you're parsing or emitting YAML, you'll likely use an options hash to tell YAML.rb how to handle data. Most YAML.rb functions can take an options hash as their final parameter. For example, the to_yaml method from the previous chapters can accept an options hash:

```
puts [[ 'Crispin', 'Glover' ]].to_yaml( :Indent => 4, :UseHeader => true, :UseVersion =>
true )
# prints:
#   --- %YAML:1.0
#   -
#       - Crispin
#       - Glover
```

Ex. 24: Using to_yaml with an options Hash

As you can see, the options hash consists of key/value pairs which affect the output of to_yaml. When parsing, many of these options are set when a YAML document is loaded. For example, when the above YAML document is loaded by YAML::load_stream, the version number and the indent will be used to set the default options for the loaded YAML::Stream object. In an options hash, the key is a symbol, selected from any of the option symbols listed below:

```
Indent: The default indentation to use when emitting (defaults to 2)
Separator: The default separator to use between documents (defaults to '---')
```

```
SortKeys: Sort Hash keys when emitting? (defaults to false)
UseHeader: Display the YAML header when emitting? (defaults to false)
UseVersion: Display the YAML version when emitting? (defaults to false)
AnchorFormat: A formatting string for anchor IDs when emitting (defaults to 'id%03d')
ExplicitTypes: Use explicit types when emitting? (defaults to false)
BestWidth: The character width to use when folding text (defaults to 80)
UseFold: Force folding of text when emitting? (defaults to false)
UseBlock: Force all text to be literal when emitting? (defaults to false)
Encoding: Unicode format to encode with (defaults to :Utf8; requires Iconv)
```

Ex. 25: Available symbols for an options Hash

See the YAML Module Reference later in this manual for any of the following functions which all can receive an options hash:

```
any Object#to_yaml method
YAML::Stream.new
YAML::Store.new
YAML::emitter_proc
```

Ex. 26: Methods which take an options Hash

2.6. Replacing PStore

PStore is a common Ruby module which serializes objects to a file. PStore is accessed as a Hash by opening a transaction with the file. YAML.rb includes YAML::Store, a drop-in replacement for PStore.

The YAML::Store class simply needs a filename to write to when it is initialized, along with any options you like:

```
require 'yaml'
y = YAML::Store.new( "/tmp/yaml.store.1", :Indent => 2 )
y.transaction do
  y['names'] = ['Crispin', 'Glover']
  y['hello'] = {'hi' => 'hello', 'yes' => 'YES!!' }
end
```

Ex. 27: Initializing YAML::Store

Like PStore, the YAML::Store class can store object hierarchies, each identified by a string. The hierarchy is store in a single YAML document as a YAML mapping.

```
hello:
  hi: hello
  yes: YES!!
names:
  - Crispin
  - Glover
```

Ex. 28: Dump of /tmp/yaml.store.1

2.7. Loading YAML Documents

YAML.rb includes a stream parser, which can read YAML from strings, files, and any type of IO. You can use YAML::load to read single documents, YAML::load_stream to read several documents at once, and YAML::load_documents to iterate through documents in a stream.

Loading a single document

Often you will want to load a single document, representing a single object, into a Ruby variable. The YAML::load method is designed to do just that. It takes either a String or an IO object and returns the first object in the document.

```
readme = YAML::load( File.open( 'README' ) )
```

Ex. 37: YAML::load Example

YAML::load is a very convenient function, as you can manipulate the YAML structure as a Ruby type. It flexes YAML's strength as a data serialization language. While an Object's to_yaml method exports it to YAML, the YAML::load method imports the Object back.

```
o = [ 'array', 'of', 'items' ]
o2 = YAML::load( o.to_yaml )
# o2 and o should be equal
```

Ex. 29: YAML::load, the answer to Object#to_yaml

Loading many documents

A YAML stream can contain more than one document. Often, you won't want to load the entire stream into memory. Rather, you'll want to load one document at a time. In Ruby, we use the YAML::load_documents method to iterate through documents.

For example, suppose we have a web server's log file, which is made up of several YAML documents in a stream:

```
---
at: 2001-08-12 09:25:00.00 Z
type: GET
HTTP: '1.0'
url: '/index.html'
---
at: 2001-08-12 09:25:10.00 Z
type: GET
HTTP: '1.0'
url: '/toc.html'
```

Ex. 39: Stream containing a log file

If we wanted to loop through the documents in this file, printing a short summary of each line, we could use YAML::load_documents:

```
require 'yaml'
log = File.open( "/var/log/apache.yaml" )
yp = YAML::load_documents( log ) { |doc|
  puts "#{doc['at']} #{doc['type']} #{doc['url']}"
}
```

Ex. 40: Loading the log file with YAML::load_documents

Like YAML::load, YAML::load_documents is called with the IO object or String that you want to read from. You also must pass YAML::load_documents a Ruby proc for handling each document. The proc only receives one parameter: the current YAML document, loaded as a Ruby object. In the example above, we receive a Hash object for each document in the stream.

YAML::load_documents is the most efficient way to load streaming data. This applies as well to TCP sockets. Client/server applications which communcate in YAML can pass the TCPSocket object directly to YAML::load_documents for parsing a stream over TCP/IP.

Loading an entire stream

In some situations, you may choose to load an entire stream for modification and re-emission. The YAML::Stream object can hold many documents and contains a few function to add convenience to editing documents in the stream. To load an entire stream into a YAML::Stream object, use the YAML::load_stream method.

Like the other YAML load functions, YAML::load_stream requires an IO object or String as its parameter:

```
readme_doc = YAML::load_stream( File.open( 'README' ) )
puts readme_doc.documents[0]['title']
# prints:
#   YAML.rb
```

Ex. 38: YAML::load_stream Example

2.8. Parsing YAML Documents

When we talk about 'loading' a YAML stream, we mean that a YAML document is translated into native types. In Ruby, this might be a Hash, an Array or any other Ruby object. But before YAML is loaded into those types, it must be parsed. Parsing is the stage where the structure of the document becomes apparent, but not the native typing.

YAML.rb gives you access to a YAML document before it is transformed. At this stage, the document is represented as a tree of YAML::YamlNode objects. This structure can be quite useful for accessing the data as a raw structure, much as the XML world has their DOM API. Also, you can use YPath queries to retrieve data from the structure. Schemas can be applied to the YamlNode tree, to validate if the structure is intact and syntactically correct.

The YAML::parse and YAML::parse_documents methods are way of accessing this parsed data.

Parsing a single document

The YAML::parse method has the same syntax as the YAML::load method. A single IO object or String containing a YAML document is passed in to the method. Rather than returning a native Ruby object, though, the YAML::parse method returns a YamlNode representing the document.

```
tree = YAML::parse( File.open( "README" ) )
puts tree.type_id
# prints:
#   map
title = tree.select( "/title" )[0]
puts title.value
# prints:
#   YAML.rb
obj_tree = tree.transform
puts obj_tree['title']
# prints:
#   YAML.rb
```

Ex. 43: Parsing a YAML document

The YamlNode returned contains type and value information for the root-level collection or scalar. If, for example, the document contains a mapping at the root level, then the YamlNode will have a type_id of 'map' and a map of YamlNodes will be contained the object's 'value' property.

```
node = YAML::parse( <<EOY )
one: 1
two: 2
EOY
puts node.type_id
# prints: 'map'
p node.value['one']
# prints key and value nodes:
#   [ #<YAML::YamlNode:0x8220278 @type_id="str", @value="one", @kind="scalar">,
#     #<YAML::YamlNode:0x821fcd8 @type_id="int", @value="1", @kind="scalar"> ]'
# Mappings can also be accessed for just the value by accessing as a Hash directly
p node['one']
# prints: #<YAML::YamlNode:0x821fcd8 @type_id="int", @value="1", @kind="scalar">
```

Ex. 30: YamlNode representing a root-level mapping

Traversing a tree of YamlNodes can be painstaking in comparison to having the native types around. YPath statements are a much quicker means of querying for the data you need. YPath queries also give you a way to build new sets of YamlNodes for transformation.

The YamlNode#select method can be used to retrieve a sequence of matching nodes. The YamlNode#transform method can be applied to a YamlNode to complete the loading of a node into a native Ruby type.

```
players = YAML::parse( <<EOY )
  player:
    - given: Sammy
      family: Sosa
    - given: Ken
      family: Griffey
    - given: Mark
      family: McGwire
EOY
given = players.select( "/player/*/given" )
p given.transform
# prints:
#   ["Sammy", "Ken", "Mark"]
```

Ex. 45: Transforming the results of a YPath selection

Parsing many documents

The YAML::parse_documents method is identical to the YAML::load_documents method, except that the iterator loops through each document returning a YamlNode for that document. YPath expressions, schema validations, and transformations can all be applied to this YamlNode, as described above.

```
require 'yaml'
log = File.open( "/var/log/apache.yaml" )
yp = YAML::parse_documents( log ) { |tree|
  at = tree.select('/at')[0].value
  type = tree.select('/type')[0].value
  puts "#{at} #{type}"
}
```

Ex. 44: Parsing YAML documents from a stream

2.9. Type Families

Typing is YAML's most extensible feature. YAML.rb is setup to handle Ruby objects under the ruby.yaml.org domain. When a document is loaded containing an object flagged with the type !ruby/symbol, the parser knows to handle this object inside of YAML.rb. Other programming languages likely don't have handlers for !ruby types. You may want to create your own type families to pass to other languages.

Loading a Type Family with a Domain

Your type families must be classified with a domain. For example, if you were the owner of 'rubyjunkies.com', then might want to categorize your types under that domain. A YAML document using your own custom address book could look like this:

```
--- !rubyjunkies.com,2002-10-24/addressBook
name: Bunbury Olsen
phone: 801-090-0900
```

```
address: |
  12 E. 400 S.
  SLC, UT 84020
```

Ex. 31: Custom type family assigned to a domain

A type family can be loaded however you choose. Using the YAML.add_domain_type method, you can register a new type, such as the address book type above:

```
YAML.add_domain_type( "rubyjunkies.com,2002-10-24", "addressBook" ) { |type, val|
  # Do something with 'val' here
}
```

Ex. 32: Registering the address book type with YAML.add_domain_type

In most cases, you'll want to map a type family directly to a class. Your address book type likely has an AddressBook class counterpart with 'name', 'phone', and 'address' attributes. The YAML.object_maker method can be used to automate your type family handler.

```
class AddressBook
  attr_accessor :name, :phone, :address
end
YAML.add_domain_type( "rubyjunkies.com,2002-10-24", "addressBook" ) { |type, val|
  YAML.object_maker( AddressBook, val )
}
```

Ex. 33: YAML.add_domain_type and YAML.object_maker

Emitting a Type Family with a Domain

So you're now covered to load your custom YAML type into a class. But you'll also need a to_yaml method to emit this class under your domain type. The default to_yaml method for the AddressBook class will emit as type family "!ruby/Object:AddressBook". To emit as type family "!rubyjunkies.com,2002-10-24/addressBook", you can overload the to_yaml method:

```
class AddressBook
  def to_yaml( opts = {} )
    YAML.quick_emit( self.id, opts ) { |out|
      out.map( "!rubyjunkies.com,2002-10-24" ) { |map|
        instance_variables.sort.each { |iv|
          map.add( iv[1..-1], instance_eval( iv ) )
        }
      }
    }
  end
end
```

Ex. 34: Overloading to_yaml for your type family

The above code is quite verbose and we'll go into a quicker technique in the next example. But this example does illustrate a few other methods which are at you disposal in customizing the to_yaml method.

The YAML.quick_emit method takes two parameters: the Object id and the options hash. The id helps YAML.rb detect duplicates in the stream, supplying an anchor for the duplicated object. If you don't want to use an anchor, merely pass nil in as the id.

For most objects, though you may just want to emit a custom type name. In this case, merely overload the to_yaml_type method in your class.

```
def to_yaml_type
  "!rubyjunkies.com,2002-10-24/addressBook"
end
```

Ex. 35: Quicker to_yaml_type for classes

Another popular request among developers who use object type families concerns ordering of properties. Not only ordering the properties displayed in the YAML document, but also suppressing properties.

By overloading the to_yaml_properties method, you can control which properties are emitted and in which order. The to_yaml_properties method should return an array of property names, along with their '@' prefix.

To order the @name, @phone, and @address properties:

```
def to_yaml_properties
  [ '@name', '@phone', '@address' ]
end
```

Ex. 36: Overloading to_yaml_properties

2.10. For More Information

YAML.rb is still under heavy development, with new updates every week. You can hear about the latest developments on the YAML mailing list [http://lists.sourceforge.net/lists/listinfo/yaml-core]. New releases are announced on the YAML.rb website [http://yaml4r.sourceforge.net/].

```
- YAML:
  - Primary site: http://www.yaml.org/
  - WikiWiki: http://wiki.yaml.org/yamlwiki/
  - IRC: irc:irc.openprojects.net/yaml
  - Specification: http://www.yaml.org/spec/
  - Productions: http://helide.com/g/yaml/yaml-productions.htm
- YAML.rb:
  - Primary site: http://yaml4r.sf.net/
  - WikiWiki: http://wiki.yaml.org/yamlwiki/YamlForRuby
  - Docs: http://yaml4r.sf.net/doc/
  - Cookbook: http://yaml4r.sf.net/cookbook/
  - Project Page: http://sf.net/projects/yaml4r/
- YAML.pm (Perl):
  - Primary site: http://search.cpan.org/search?query=YAML&mode=all
- PyYaml (Python):
  - WikiWiki: http://wiki.yaml.org/yamlwiki/PurePythonParserForYaml
  - Debian packages: http://files.zefamily.org/debian
- YAMLj (Java):
  - Primary site: http://helide.com/g/yaml/
```

Ex. : YAML on the Web

3. Reference

3.1. YAML Module

3.1.1. YAML::add_domain_type Method

Adds a user-level domain type to the parser

```
YAML::add_domain_type(
  (String) domain_and_date,
  (Regexp or String) type_re,
  (Proc) transfer_proc
)
```

Parameters

domain_and_date

The domain and date (seperated by a comma) to assign the type under. An example for a personal type would be 'your-company.com,2002-09-23'. The date can usually be just a year, representing the first day of year ('2002' == '2002-01-01'). First day of the month can be shortened to just year and month ('2002-09' == '2002-09-01'). See http://www.taguri.org/ for details on this convention.

type_re

A regular expression to match type names with. If a String, the exact name of the type to add.

transfer_proc

A procedure for translating the YAML element into the domain type.

Block Parameters

type

The full domain type string of the element being parsed.

val

The value of the YAML element being coerced into this domain type.

Returns

None

Details

The add_domain_type method allows you to register your own domain-specific types to YAML's typing mechanism. The domain string should contain a tag-uri domain (with the domain name and date separated by a comma).

```
YAML.add_domain_type( "hospital.com,2003", "Med" ) do |type, val|
  Medication.new( val )
end
```

Ex. 46: Adding a Domain Type

3.1.2. YAML::add_private_type Method

Adds a user-level private type to the parser

```
YAML::add_private_type(
  (Regexp or String) type_re,
  (Proc) transfer_proc
)
```

Parameters

type_re

A regular expression to match type names with. If a String, the exact name of the type to add.

transfer_proc

A procedure for translating the YAML element into the private type.

Block Parameters

type

The type string of the element being parsed.

val

The value of the YAML element being coerced into this private type.

Returns

None

Details

Private types are intended to be a quick and informal typing mechanism. If, for example, you have a Hash that is storing employee data, but you want to be able to mark the Hash as containing structured employee data, you could give the Hash an '!!EmployeeList' private type, without needing any type of Ruby internal type to back it up.

```
YAML.add_domain_type( "hospital.com,2003", "Med" ) do |type, val|
    Medication.new( val )
  end
```

Ex. 46: Adding a Domain Type

Often, for simple private types, you don't even need to use the YAML::add_private_type method. Any private types found by the parser which aren't registered become objects of the PrivateType class, another convenient way of handling these special creatures.

3.1.3. YAML::load Method

Loads a single document from a stream

```
YAML::load(
  (String or IO) io
)
```

Parameters

io

The string or IO object to read from.

Returns

A Ruby object

Details

The YAML::load method is for quick access to files containing a single YAML document. The document is parsed and the object it contains is returned. For example, to load the README that comes with YAML.rb:

```
readme = YAML::load( File.open( 'README' ) )
```

Ex. 37: YAML::load Example

Since this method only parses a single document, the IO object is closed when the method exits.

3.1.4. YAML::load_documents Method

Iterates through documents in a stream

```
YAML::load_documents(
  (String or IO) io,
  (Proc) doc_proc
)
```

Parameters

io

The string or IO object to read from.

doc_proc

A procedure for handling each parsed document

Block Parameters

obj

An object containing the current document

Returns

A Ruby object

Details

The YAML::load_documents method is great for parsing streaming data, especially data which has a fixed formatting. For example, let's suppose you are reading from a log file:

```
---
at: 2001-08-12 09:25:00.00 Z
type: GET
HTTP: '1.0'
url: '/index.html'
---
at: 2001-08-12 09:25:10.00 Z
type: GET
HTTP: '1.0'
url: '/toc.html'
```

Ex. 39: Stream containing a log file

Using YAML::load_documents, you can process each entry individually, without needing to allocate space for the entire file contents in memory. With each iteration, the current document is passed into the Proc you supply:

```
require 'yaml'
log = File.open( "/var/log/apache.yaml" )
yp = YAML::load_documents( log ) { |doc|
  puts "#{doc['at']} #{doc['type']} #{doc['url']}"
}
```

Ex. 40: Loading the log file with YAML::load_documents

The IO object is closed upon completion of parsing.

3.1.5. YAML::load_stream Method

Loads an entire YAML stream into a new YAML::Stream object.

```
YAML::load_stream(
  (String or IO) io
)
```

Parameters

io

The string or IO object to read from.

Returns

A YAML::Stream object

Details

The YAML::load_stream method will iterate through the documents in a YAML stream, building them up inside of a YAML::Stream object:

```
readme_doc = YAML::load_stream( File.open( 'README' ) )
puts readme_doc.documents[0]['title']
# prints:
#   YAML.rb
```

Ex. 38: YAML::load_stream Example

Any options within the YAML stream are preserved in the YAML::Stream object and the IO object is closed upon completion of parsing.

3.1.6. YAML::parse Method

Loads a single document as a YamlNode tree

```
YAML::parse(
  (String or IO) io
)
```

Parameters

io

The string or IO object to read from.

Returns

A YamlNode object or nil if no document found.

Details

The YAML::parse method loads a single YAML document from a stream into a YamlNode. The YamlNode can be used to apply YPath expressions or validate against a schema structure.

```
tree = YAML::parse( File.open( "README" ) )
puts tree.type_id
# prints:
#   map
title = tree.select( "/title" )[0]
puts title.value
# prints:
#   YAML.rb
obj_tree = tree.transform
puts obj_tree['title']
# prints:
#   YAML.rb
```

Ex. 43: Parsing a YAML document

With the YamlNode, you can access data before it's typed and transformed into Ruby native types. A tree of YamlNodes can later be turned into Ruby native types by using the YamlNode#transform method.

3.1.7. YAML::parse_documents Method

Iterates through documents in a stream, returning YamlNodes for each

```
YAML::parse_documents(
  (String or IO) io,
  (Proc) doc_proc
)
```

Parameters

io

The string or IO object to read from.

doc_proc

A procedure for handling each parsed document

Block Parameters

obj

An object containing the current document as a YamlNode

Returns

Nil

Details

Just as YAML::parse provides access to the generic data of a document, YAML::parse_documents iterates through documents in a stream, providing a tree of YamlNodes for you to work on. For example, let's suppose you are reading from a log file:

```
---
at: 2001-08-12 09:25:00.00 Z
type: GET
HTTP: '1.0'
url: '/index.html'
---
at: 2001-08-12 09:25:10.00 Z
type: GET
HTTP: '1.0'
url: '/toc.html'
```

Ex. 39: Stream containing a log file

Using YAML::parse_documents, you can process each entry in a file individually, without needing to allocate space for the entire file contents in memory. With each iteration, the current document is passed into the Proc you supply. From there, YPath expressions or transformations could be applied:

```
require 'yaml'
log = File.open( "/var/log/apache.yaml" )
yp = YAML::parse_documents( log ) { |tree|
  at = tree.select('/at')[0].value
  type = tree.select('/type')[0].value
  puts "#{at} #{type}"
}
```

Ex. 44: Parsing YAML documents from a stream

The IO object is closed upon completion of parsing.

3.1.8. YAML::Stream Class

3.1.8.1. YAML::Stream#new Method

Creates a new Stream object

```
aYamlStream.new(
  (Hash) opts
)
```

Parameters

opts

An option hash

Returns

None

Details

The YAML::Stream object is a simple means of organizing many YAML documents into a single stream. A Stream object can be created with an option hash or, alternatively, loaded by YAML::load_stream.

```
d = YAML::Stream.new( :Indent => 4, :UseHeader => true )
d.add( 'one' )
d.add( 'two' )
d.add( 'three' )
puts d.emit
# prints:
#   --- one
#   --- two
#   --- three
```

Ex. 42: YAML::Stream.new

3.1.8.2. YAML::Stream#add Method

Appends a new document to the Stream.

```
aYamlStream.add(
  (Object) doc
)
```

Parameters

doc

After addition, this document will appear at the end of the YAML stream.

Returns

None

3.1.8.3. YAML::Stream#edit Method

Replaces a document at the given index in the Stream object.

```
aYamlStream.edit(
  (Integer) doc_num,
  (Object) doc
)
```

Parameters

doc_num

The index in the @documents array to place this object.

doc

The Object to place in the Stream.

Returns

None

3.1.8.4. YAML::Stream#emit Method

Emits this Stream as YAML.

```
aYamlStream.emit()
```

Parameters

None

Returns

None

3.1.9. YAML::Emitter Class

3.1.9.1. YAML::Emitter#new Method

Creates a new Emitter object

```
aYamlEmitter.new()
```

Parameters

None

Returns

None

3.1.10. YAML::Parser Class

3.1.10.1. YAML::Parser#new Method

Creates a Parser object.

```
aYamlParser.new()
```

Parameters

None

Returns

None

Details

The Parser is written largely using a Racc LALR grammar.

3.1.10.2. YAML::Parser#parse Method

Loads a single object from a YAML stream.

```
aYamlParser.parse(
  (String or IO) io
)
```

Parameters

io

The string or IO object to read from.

Returns

A Ruby object

Details

This method is used by YAML::load to load a single document. In fact, YAML::load is equivalent to:

```
YAML::Parser.new.parse( io )
```

Ex. 41: YAML.load Source

3.1.10.3. YAML::Parser#parse_documents Method

Iterates through objects in a YAML stream.

```
aYamlParser.parse_documents(
  (String or IO) io,
  (Proc) doc_proc
)
```

Parameters

io

The string or IO object to read from.

doc_proc

A procedure for handling each parsed document

Block Parameters

obj

An object containing the current document

Returns

None

Details

This method is used by YAML::load_documents, YAML::parse_documents, and YAML::load_stream

3.1.11. YAML::Store Class

3.1.11.1. YAML::Store#new Method

Creates a new Store object.

```
aYamlStore.new(
  (String) file,
  (Hash) opts
)
```

Parameters

file

Name of the YAML::Store file to create, write to and read from.

opts

An option hash

Returns

None

3.1.11.2. YAML::Store#transaction Method

Opens a transaction with the YAML::Store file.

```
aYamlStore.transaction()
```

Parameters

None

Block Parameters

doc

The Hash representing named object hierarchies.

Returns

None

3.1.12. YAML::YamlNode Class

3.1.12.1. YAML::YamlNode#new Method

Creates a new YamlNode object.

```
aYamlYamlNode.new(
  (String) type_id,
  (Object) value
)
```

Parameters

type_id

The transfer method attached to this node.

value

The generic data to store in this node.

Returns

None

3.1.12.2. YAML::YamlNode#emit Method

Transforms this node and returns a YAML dump of the object.

```
aYamlYamlNode.emit()
```

Parameters

None

Returns

String

Details

This method is simply an alias for aYamlNode.transform.to_yaml.

3.1.12.3. YAML::YamlNode#search Method

Performs a YPath search, returning qualified paths.

```
aYamlYamlNode.search(
  (String) ypath
)
```

Parameters

ypath

The YPath statement.

Returns

An Array of Strings or nil if none qualified.

Details

This method is EXPERIMENTAL and much of the YPath syntax isn't supported. YPath is still being defined by YAML implementors and this is all subject to change.

3.1.12.4. YAML::YamlNode#select Method

Performs a YPath search, returning qualified nodes.

```
aYamlYamlNode.select(
  (String) ypath
)
```

Parameters

ypath

The YPath statement.

Returns

A YamlNode or nil if none qualified.

Details

This method is EXPERIMENTAL and much of the YPath syntax isn't supported. YPath is still being defined by YAML implementors and this is all subject to change.

3.1.12.5. YAML::YamlNode#transform Method

Applies transfer methods to this YamlNode and its children.

```
aYamlYamlNode.transform()
```

Parameters

None

Returns

A Ruby object.

Details

The YamlNode#transform method is used to turn a tree of YamlNodes into a native Ruby object, as you would expect from a loading method, such as YAML::load. This method is handy if you want to perform a YPath select to grab a group of nodes and turn them into a new document.

```
players = YAML::parse( <<EOY )
  player:
    - given: Sammy
      family: Sosa
    - given: Ken
      family: Griffey
    - given: Mark
      family: McGwire
EOY
given = players.select( "/player/*/given" )
p given.transform
# prints:
#   ["Sammy", "Ken", "Mark"]
```

Ex. 45: Transforming the results of a YPath selection

4. Examples

This page contains all of the examples from throughout this documentation. Convenient? Perhaps?

```
- dogs
- cats
- badgers
```

Ex. 1: Simple sequence in YAML

```
[ 'dogs', 'cats', 'badgers' ]
```

Ex. 2: Simple sequence in Ruby

```
require 'yaml'
puts( [ 'dogs', 'cats', 'badgers' ].to_yaml )
# prints:
#   - dogs
#   - cats
#   - badgers
```

Ex. 3: Using to_yaml to output a sequence

```
-
  - pineapple
  - coconut
-
  - umbrella
  - raincoat
```

Ex. 4: Nested sequence in YAML

```
dog: canine
cat: feline
badger: malign
```

Ex. 5: Simple mapping in YAML

```
{ 'dog' => 'canine',
  'cat' => 'feline',
  'badger' => 'malign' }
```

Ex. 6: Simple mapping in Ruby

```
require 'yaml'
puts( { 'dog' => 'canine',
        'cat' => 'feline',
        'badger' => 'malign' }.to_yaml )
# prints:
#   dog: canine
#   cat: feline
#   badger: malign
```

Ex. 7: Using to_yaml to output a mapping

```
Joey:
  age: 22
  sex: M
Laura:
  age: 24
  sex: F
```

Ex. 8: Nested mapping

```
- ~
- true
- 10
- 10.2
- 2002-08-15T17:18:23.18-06:00
- 2002-08-15
```

Ex. 9: Basic types in YAML

```
~: NilClass
+: TrueClass
true: TrueClass
True: TrueClass
-: FalseClass
false: FalseClass
False: FalseClass
0: Integer
1: Integer
100: Integer
1,000: Integer
0.0: Float
1.0: Float
100.001: Float
1,000.001: Float
1.00009e+3: Float
2002-08-15T17:18:23.18-06:00: Time
2002-08-15 17:18:23.18 -06:00: Time
1976-07-31: Date
```

Ex. 10: Basic types and their Ruby classes

```
puts( nil.to_yaml )
# prints:
#   --- ~
puts( true.to_yaml )
# prints:
#   --- true
```

```
puts( false.to_yaml )
# prints:
#   --- (false)
puts( 10.to_yaml )
# prints:
#   --- 10
puts( 20.45.to_yaml )
# prints:
#   --- 20.45
puts( Time.now.to_yaml )
# prints:
#   --- 2002-08-15T17:29:01.79-06:00
puts( Date.new( 1976, 07, 31 ).to_yaml )
# prints:
#   --- 1976-07-31
```

Ex. 11: Using to_yaml with basic types

```
- literal: |
    A literal block keeps all
    new lines when it is brought
    into Ruby.
- folded: >
    A folded block will get rid
    of its newlines, trading them
    for spaces when it is brought
    into Ruby.
```

Ex. 12: Literal and folded blocks

```
txt = "Just got my (dead-tree, printed-on-paper, I don't know if there's a web " +
      "version) copy of Linux Magazine for September, 2002.  There's an article by " +
      "Dave Thomas about building networked applications in Ruby.\n\nProps to Dave!"
puts txt.to_yaml
# prints:
#   --- >-
#   Just got my (dead-tree, printed-on-paper, I don't know if there's a web version)
#   copy of Linux Magazine for September, 2002.  There's an article by Dave Thomas
#   about building networked applications in Ruby.
#
#
#   Props to Dave!
```

Ex. 13: Case where to_yaml generates a folded block

```
txt =<<EOF
ZenWeb 2.11.0 has been released!
I don't think I've remembered to announce any releases in a while, so
this one is a tad different. Among the newest changes are:
  + relative url renderer
  + massively improved demo/docs
:)
EOF
puts txt.to_yaml
# prints:
#   --- |
#
#   ZenWeb 2.11.0 has been released!
#
#   I don't think I've remembered to announce any releases in a while, so
#   this one is a tad different. Among the newest changes are:
#
#     + relative url renderer
#     + massively improved demo/docs
#
#   :)
#
```

Ex. 14: Case where to_yaml generates a literal block

```
simple symbol: !ruby/symbol Simple
shortcut syntax: !ruby/sym Simple
symbols in maps:
  !ruby/sym MapKey: !ruby/sym MapValue
```

Ex. 15: Symbols in YAML

```
puts :Simple.to_yaml
# prints:
#   --- !ruby/sym Simple
```

Ex. 16: Using to_yaml with Symbols

```
normal range: !ruby/range 10..20
exclusive range: !ruby/range 11...20
negative range: !ruby/range -1..-5
? !ruby/range 0..40
: range as a map key
```

Ex. 17: Ranges in YAML

```
floats: !ruby/range 10.2..21.5
dates: !ruby/range 2001-03-02..2001-09-11
timestamps: !ruby/range 2001-09-03T05:16:23Z..2001-09-04T07:08:13Z
```

Ex. 18: Other basic types in Ranges

```
starts with a b: !ruby/regexp '/^b/'
ends with a z: !ruby/regexp '/z$/'
search for a C or c: !ruby/regexp '/c/i'
```

Ex. 19: Regexps in YAML

```
puts( /^b/.to_yaml )
# prints:
#   --- !ruby/regexp "/^b/"
```

Ex. 20: Using to_yaml with Regexps

```
class Video
  attr_accessor :title, :year, :rating
  def initialize( t, y, r )
    @title = t
    @year = y
    @rating = r
  end
end
```

Ex. 21: Video class

```
collection = [
  Video.new( "Ishtar", 1987, 8.8 ),
  Video.new( "Dr. Strangelove", 1964, 10.0 )
]
```

Ex. 22: Your Video collection

```
puts collection.to_yaml
# prints:
#   - !ruby/object:Video
#     title: Ishtar
#     year: 1987
#     rating: 8.8
#   - !ruby/object:Video
#     title: Dr. Strangelove
#     year: 1964
#     rating: 10.0
```

Ex. 23: Your Videos in YAML

```
puts [[ 'Crispin', 'Glover' ]].to_yaml( :Indent => 4, :UseHeader => true, :UseVersion =>
true )
# prints:
#   --- %YAML:1.0
#   -
#       - Crispin
#       - Glover
```

Ex. 24: Using to_yaml with an options Hash

```
Indent: The default indentation to use when emitting (defaults to 2)
Separator: The default separator to use between documents (defaults to '---')
```

```
SortKeys: Sort Hash keys when emitting? (defaults to false)
UseHeader: Display the YAML header when emitting? (defaults to false)
UseVersion: Display the YAML version when emitting? (defaults to false)
AnchorFormat: A formatting string for anchor IDs when emitting (defaults to 'id%03d')
ExplicitTypes: Use explicit types when emitting? (defaults to false)
BestWidth: The character width to use when folding text (defaults to 80)
UseFold: Force folding of text when emitting? (defaults to false)
UseBlock: Force all text to be literal when emitting? (defaults to false)
Encoding: Unicode format to encode with (defaults to :Utf8; requires Iconv)
```

Ex. 25: Available symbols for an options Hash

```
any Object#to_yaml method
YAML::Stream.new
YAML::Store.new
YAML::emitter_proc
```

Ex. 26: Methods which take an options Hash

```
require 'yaml'
y = YAML::Store.new( "/tmp/yaml.store.1", :Indent => 2 )
y.transaction do
  y['names'] = ['Crispin', 'Glover']
  y['hello'] = {'hi' => 'hello', 'yes' => 'YES!!' }
end
```

Ex. 27: Initializing YAML::Store

```
hello:
  hi: hello
  yes: YES!!
names:
  - Crispin
  - Glover
```

Ex. 28: Dump of /tmp/yaml.store.1

```
o = [ 'array', 'of', 'items' ]
o2 = YAML::load( o.to_yaml )
# o2 and o should be equal
```

Ex. 29: YAML::load, the answer to Object#to_yaml

```
node = YAML::parse( <<EOY )
one: 1
two: 2
EOY
puts node.type_id
# prints: 'map'
p node.value['one']
# prints key and value nodes:
```

```
#   [ #<YAML::YamlNode:0x8220278 @type_id="str", @value="one", @kind="scalar">,
#     #<YAML::YamlNode:0x821fcd8 @type_id="int", @value="1", @kind="scalar"> ]'
# Mappings can also be accessed for just the value by accessing as a Hash directly
p node['one']
# prints: #<YAML::YamlNode:0x821fcd8 @type_id="int", @value="1", @kind="scalar">
```

Ex. 30: YamlNode representing a root-level mapping

```
--- !rubyjunkies.com,2002-10-24/addressBook
name: Bunbury Olsen
phone: 801-090-0900
address: |
  12 E. 400 S.
  SLC, UT 84020
```

Ex. 31: Custom type family assigned to a domain

```
YAML.add_domain_type( "rubyjunkies.com,2002-10-24", "addressBook" ) { |type, val|
  # Do something with 'val' here
}
```

Ex. 32: Registering the address book type with YAML.add_domain_type

```
class AddressBook
  attr_accessor :name, :phone, :address
end
YAML.add_domain_type( "rubyjunkies.com,2002-10-24", "addressBook" ) { |type, val|
  YAML.object_maker( AddressBook, val )
}
```

Ex. 33: YAML.add_domain_type and YAML.object_maker

```
class AddressBook
  def to_yaml( opts = {} )
    YAML.quick_emit( self.id, opts ) { |out|
      out.map( "!rubyjunkies.com,2002-10-24" ) { |map|
        instance_variables.sort.each { |iv|
          map.add( iv[1..-1], instance_eval( iv ) )
        }
      }
    }
  end
end
```

Ex. 34: Overloading to_yaml for your type family

```
def to_yaml_type
  "!rubyjunkies.com,2002-10-24/addressBook"
end
```

Ex. 35: Quicker to_yaml_type for classes

```
def to_yaml_properties
  [ '@name', '@phone', '@address' ]
end
```

Ex. 36: Overloading to_yaml_properties

```
readme = YAML::load( File.open( 'README' ) )
```

Ex. 37: YAML::load Example

```
readme_doc = YAML::load_stream( File.open( 'README' ) )
puts readme_doc.documents[0]['title']
# prints:
#   YAML.rb
```

Ex. 38: YAML::load_stream Example

```
---
at: 2001-08-12 09:25:00.00 Z
type: GET
HTTP: '1.0'
url: '/index.html'
---
at: 2001-08-12 09:25:10.00 Z
type: GET
HTTP: '1.0'
url: '/toc.html'
```

Ex. 39: Stream containing a log file

```
require 'yaml'
log = File.open( "/var/log/apache.yaml" )
yp = YAML::load_documents( log ) { |doc|
  puts "#{doc['at']} #{doc['type']} #{doc['url']}"
}
```

Ex. 40: Loading the log file with YAML::load_documents

```
YAML::Parser.new.parse( io )
```

Ex. 41: YAML.load Source

```
d = YAML::Stream.new( :Indent => 4, :UseHeader => true )
d.add( 'one' )
d.add( 'two' )
d.add( 'three' )
puts d.emit
# prints:
#   --- one
#   --- two
#   --- three
```

Ex. 42: YAML::Stream.new

```
tree = YAML::parse( File.open( "README" ) )
puts tree.type_id
# prints:
#   map
title = tree.select( "/title" )[0]
puts title.value
# prints:
#   YAML.rb
obj_tree = tree.transform
puts obj_tree['title']
# prints:
#   YAML.rb
```

Ex. 43: Parsing a YAML document

```
require 'yaml'
log = File.open( "/var/log/apache.yaml" )
yp = YAML::parse_documents( log ) { |tree|
  at = tree.select('/at')[0].value
  type = tree.select('/type')[0].value
  puts "#{at} #{type}"
}
```

Ex. 44: Parsing YAML documents from a stream

```
players = YAML::parse( <<EOY )
  player:
    - given: Sammy
      family: Sosa
    - given: Ken
      family: Griffey
    - given: Mark
      family: McGwire
EOY
given = players.select( "/player/*/given" )
p given.transform
# prints:
#   ["Sammy", "Ken", "Mark"]
```

Ex. 45: Transforming the results of a YPath selection

```
YAML.add_domain_type( "hospital.com,2003", "Med" ) do |type, val|
  Medication.new( val )
end
```

Ex. 46: Adding a Domain Type