Kong Documentation

Release 0.1.0

Hong Minhee

August 18, 2014

Contents

Contents:

CHAPTER 1

kong — Tofu implementation

1.1 kong.ast — Abstract Syntax Tree

class kong.ast.**Node**
Bases: object

An abstract base class for syntax tree.

class kong.ast.**ExpressionList**
Bases: tuple

An abstract base class for Expression list.

class kong.ast.**Program**
Bases: kong.ast.Node, kong.ast.ExpressionList

A program node.

```
program    ::=   (expr terminate)* [expr]
terminate  ::=   (";" | newline)+
newline    ::=   ["\r"] "\n"
```

```
>>> Program([Identifier(u'abc')])
kong.ast.Program([kong.ast.Identifier(u'abc')])
>>> print unicode(_)
abc
```

Parameters **expressions** (collections.Iterable) – Expression list

class kong.ast.**Expression**
Bases: kong.ast.Node

An expression node. It is an abstract class.

```
expr  ::=   "(" expr ")" | literal | id | attr |
```

class kong.ast.**Identifier**
Bases: kong.ast.Expression, unicode

An identifier node.

```
id  ::=   /[^[:digit:][:space:]][^[:space:]]*/ except "<-"
```

Parameters **identifier** – an identifier string

class kong.ast.**Application**(*function*, *arguments*)

Bases: kong.ast.Expression

An application (call) node.

```
apply  ::=   expr "(" args ")" | expr args
args   ::=   (expr ",")* [expr]
```

```
>>> app = Application(Identifier('func'),
...                   [Identifier('a'), Identifier('b')])
>>> app
kong.ast.Application(kong.ast.Identifier(u'func'),
                     [kong.ast.Identifier(u'a'),
                      kong.ast.Identifier(u'b')])
>>> print unicode(app)
func(a, b)
```

Parameters

- **function** (Expression) – a function to apply
- **arguments** (collections.Iterable) – a Expresion list

class kong.ast.**Attribute**(*function*, *arguments=None*, *attribute=None*)

Bases: kong.ast.Application

A pseudo-attribute node.

```
attr  ::=   expr "."  (id | number)
```

```
>>> attr = Attribute(Identifier('obj'), attribute=Identifier('attr'))
>>> attr
kong.ast.Attribute(kong.ast.Identifier(u'obj'),
                   attribute=kong.ast.Identifier(u'attr'))
>>> print unicode(attr)
obj.attr
```

Parameters

- **function** (Expression) – a function to apply
- **arguments** (collections.Iterable) – a Expresion list
- **attribute** (Identifier, numbers.Integral) – an attribute name

Arguments attribute and arguments are exclusive but one of them is required.

attribute

(Identifier, numbers.Integral) Attribute name.

```
>>> attr = Attribute(Identifier('a'), [StringLiteral(12)])
>>> attr.attribute
12
>>> attr2 = Attribute(Identifier('a'), [StringLiteral(u'b')])
>>> attr2.attribute
kong.ast.Identifier(u'b')
```

class kong.ast.**Operator**(*function=None*, *arguments=None*, *operator=None*, *operands=None*)

Bases: kong.ast.Application

A pseudo-operator node.

```
operator  ::=  expr id expr
```

```
>>> op = Operator(operator=Identifier('+'),
...               operands=[StringLiteral(1), StringLiteral(2)])
>>> op.function
kong.ast.Attribute(kong.ast.StringLiteral(1),
                   attribute=kong.ast.Identifier(u'+'))
>>> op.arguments
(kong.ast.StringLiteral(2),)
>>> print unicode(op)
1 + 2
```

There are two signatures. One is the same to Application's:

Parameters

- **function** (Expression) – a function to apply
- **arguments** (collections.Iterable) – a Expresion list

Other one takes operator and operands (by keywords only):

Parameters

- **operator** (Identifier) – an operator name
- **operands** (collections.Iterable) – pair of Expression

operator

(Identifier) Operator name.

```
>>> op = Operator(Attribute(Identifier('a'),
...                         attribute=Identifier('-')),
...               [StringLiteral(123)])
>>> op.operator
kong.ast.Identifier(u'-')
```

operands

(tuple) Pair of two operands.

```
>>> op = Operator(Attribute(Identifier('a'),
...                         attribute=Identifier('-')),
...               [StringLiteral(123)])
>>> op.operands
(kong.ast.Identifier(u'a'), kong.ast.StringLiteral(123))
```

class kong.ast.**Definition**

Bases: kong.ast.Expression

An abstract class for definition nodes.

```
define  ::=   lvalue "<-" expr
lvalue  ::=   ["."]  id | attr
```

lvalue = NotImplemented
(Identifier, Attribute) Lvalue expression.

rvalue = NotImplemented
(Expression) Rvalue expression.

class kong.ast.**IdentifierDefinition**(*lvalue*, *rvalue*)
Bases: kong.ast.Definition

An abstract class for identifier definition nodes.

class kong.ast.**IdentifierLocalDefinition**(*lvalue*, *rvalue*)
Bases: kong.ast.IdentifierDefinition

Local identifier definition node.

```
>>> set = IdentifierLocalDefinition(lvalue=Identifier('abc'),
...                                 rvalue=Identifier('def'))
>>> print unicode(set)
abc <- def
```

Parameters

- **lvalue** (Identifier) – lvalue identifier
- **rvalue** (Expression) – rvalue expression

class kong.ast.**IdentifierAssignment**(*lvalue*, *rvalue*)
Bases: kong.ast.IdentifierDefinition

Identifier assignment node.

```
>>> set = IdentifierAssignment(lvalue=Identifier('abc'),
...                            rvalue=Identifier('def'))
>>> print unicode(set)
.abc <- def
```

Parameters

- **lvalue** (Identifier) – lvalue identifier
- **rvalue** (Expression) – rvalue expression

class kong.ast.**AttributeDefinition**(*function=None*, *arguments=None*, *lvalue=None*, *rvalue=None*)
Bases: kong.ast.Definition, kong.ast.Application

A definition node of attribute. Attribute definitions are just two arguments application under the hood. For example, following two expressions are equivalent:

```
obj.attr = value
obj('attr', value)
```

```
>>> attr = Attribute(Identifier('abc'), attribute=Identifier('def'))
>>> set = AttributeDefinition(lvalue=attr, rvalue=StringLiteral(123))
>>> set.function
kong.ast.Identifier(u'abc')
>>> set.arguments
(kong.ast.StringLiteral(u'def'), kong.ast.StringLiteral(123))
>>> print unicode(set)
abc.def <- 123
```

There are two signatures. One is the same to `Application`'s:

Parameters

- **function** (`Expression`) – a function to apply
- **arguments** (`collections.Iterable`) – a `Expresion` list

Other one is the same to `Definition` or `Assignment`'s (but only by keywords):

Parameters

- **lvalue** (`Attribute`, `Application`) – lvalue attribute
- **rvalue** (`Expression`) – rvalue expression

lvalue

(`Attribute`) Lvalue attribute.

```
>>> args = StringLiteral(u'attr'), StringLiteral(u'value')
>>> set = AttributeDefinition(Identifier('obj'), args)
>>> set.lvalue
kong.ast.Attribute(kong.ast.Identifier(u'obj'),
                   attribute=kong.ast.Identifier(u'attr'))
```

rvalue

(`Expression`) Rvalue expression.

```
>>> args = StringLiteral(u'attr'), StringLiteral(u'value')
>>> set = AttributeDefinition(Identifier('obj'), args)
>>> set.rvalue
kong.ast.StringLiteral(u'value')
```

class kong.ast.**Literal**

Bases: `kong.ast.Expression`

A literal node. It is an abstract class.

```
literal  ::=  str_literal | dict_literal | func_def | list_literal
```

class kong.ast.**ListLiteral**

Bases: `kong.ast.Literal`, `kong.ast.ExpressionList`

A list literal node.

```
list_literal  ::=  "[" (expr ",")* [expr] "]"
```

Parameters **expressions** (`collections.Iterable`) – `Expression` list

class kong.ast.**DictionaryLiteral**(*program*)

Bases: kong.ast.Literal

A dictionary literal node.

```
dict_literal  ::=  "{" program "}"
```

```
>>> prog = Program([
...     IdentifierLocalDefinition(Identifier('a'), StringLiteral(123)),
...     IdentifierLocalDefinition(Identifier('b'), StringLiteral(456))
... ])
>>> d = DictionaryLiteral(prog)
>>> print unicode(d)
{ a <- 123; b <- 456 }
>>> print unicode(DictionaryLiteral([]))
{}
```

Parameters **program** (Program, ExpressionList, collections.Iterable) – Expression list

class kong.ast.**FunctionLiteral**(*parameters*, *program*)

Bases: kong.ast.Literal

A function literal node.

```
func_def  ::=  "(" params ")" ":"  dict_literal
params    ::=  (id ",")* [id]
```

```
>>> params = Identifier('a'), Identifier('b')
>>> prog = Program([Operator(operator=Identifier('+'),
...                          operands=params)])
>>> f = FunctionLiteral(params, prog)
>>> print unicode(f)
(a, b): { a + b }
```

Parameters

- **parameters** (collections.Iterable) – Identifier list
- **program** (Program, ExpressionList, collections.Iterable) – a program body Expression list

class kong.ast.**StringLiteral**(*string*)

Bases: kong.ast.Literal

A string literal node.

```
>>> s = StringLiteral(u'string literal')
>>> s
kong.ast.StringLiteral(u'string literal')
>>> print unicode(s)
"string literal"
>>> n = StringLiteral(u'123')
>>> n
kong.ast.StringLiteral(123)
>>> print unicode(n)
123
```

```
str_literal  ::=  number | /"([^"]|\.)*"/
number       ::=  digit+
digit        ::=  "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

Parameters **string** (`unicode`, `numbers.Integral`) – a string

1.2 `kong.parser` — Tofu parser

`kong.parser.`**`parse_expression`**(*tokens*)

Parses an expression.

```
>>> from kong.lexer import tokenize
>>> p = lambda s: parse_expression(tokenize(s))
>>> p('f()')
kong.ast.Application(kong.ast.Identifier(u'f'), [])
>>> p('f(a, f2())')
kong.ast.Application(kong.ast.Identifier(u'f'),
    [kong.ast.Identifier(u'a'),
     kong.ast.Application(kong.ast.Identifier(u'f2'), [])])
>>> p('b <- 1 + (a <- 2) * 3')
kong.ast.IdentifierLocalDefinition(lvalue=kong.ast.Identifier(u'b'),
 rvalue=kong.ast.Operator(operator=kong.ast.Identifier(u'*'),
  operands=[kong.ast.Operator(operator=kong.ast.Identifier(u'+'),
   operands=[kong.ast.StringLiteral(1),
    kong.ast.IdentifierLocalDefinition(lvalue=kong.ast.Identifier(u'a'),
     rvalue=kong.ast.StringLiteral(2))]),
   kong.ast.StringLiteral(3)]))
>>> p('(1 + 2).*')
kong.ast.Attribute(kong.ast.Operator(operator=kong.ast.Identifier(u'+'),
  operands=[kong.ast.StringLiteral(1), kong.ast.StringLiteral(2)]),
 attribute=kong.ast.Identifier(u'*'))
>>> p('[]')
kong.ast.ListLiteral([])
>>> p('[a, 1]')
kong.ast.ListLiteral([kong.ast.Identifier(u'a'),
                      kong.ast.StringLiteral(1)])
>>> p('[1, [2, 3], +]')
kong.ast.ListLiteral([kong.ast.StringLiteral(1),
    kong.ast.ListLiteral([kong.ast.StringLiteral(2),
        kong.ast.StringLiteral(3)]),
    kong.ast.Identifier(u'+')])
```

Parameters **tokens** (`collections.Iterable`) – tokens to parse

Returns parsed expression

Return type `kong.ast.Expression`

Raises `kong.lexer.SyntaxError` for invalid syntax

1.3 kong.lexer — Tokenizer

kong.lexer.**TOKEN_PATTERN = <_sre.SRE_Pattern object at 0x21b1700>**
The re pattern that matches to tokens.

kong.lexer.**tokenize**(*stream*)
Makes tokens from input stream.

```
>>> t = lambda s: list(tokenize(s))
>>> t(u'a<-func  (123)')
[kong.lexer.Token('identifier', u'a', 0),
 kong.lexer.Token('arrow', u'<-', 1),
 kong.lexer.Token('identifier', u'func', 3),
 kong.lexer.Token('space', u'  ', 7),
 kong.lexer.Token('parenthesis', u'(', 9),
 kong.lexer.Token('number', u'123', 10),
 kong.lexer.Token('parenthesis', u')', 13)]
```

It supports streaming as well:

```
>>> stream = [u'a(12', u'3)\nb<', u'-c * 123']
>>> t(stream)
[kong.lexer.Token('identifier', u'a', 0),
 kong.lexer.Token('parenthesis', u'(', 1),
 kong.lexer.Token('number', u'123', 2),
 kong.lexer.Token('parenthesis', u')', 5),
 kong.lexer.Token('newline', u'\n', 6),
 kong.lexer.Token('identifier', u'b', 7),
 kong.lexer.Token('arrow', u'<-', 8),
 kong.lexer.Token('identifier', u'c', 10),
 kong.lexer.Token('space', u' ', 11),
 kong.lexer.Token('identifier', u'*', 12),
 kong.lexer.Token('space', u' ', 13),
 kong.lexer.Token('number', u'123', 14)]
```

Parameters **stream** (collections.Iterable) – input stream

Returns Token list

Return type collections.Iterable

class kong.lexer.**Token**(*tag*, *string*, *offset*)
A token that contains tag, string and offset.

tag
(basestring) The type of token e.g. 'arrow', 'colon'.

string
(basestring) The token string.

offset
(numbers.Integral) The token offset.

get_syntax_error(*message=None*)
Makes a SyntaxError with its offset.

Parameters **message** (basestring) – an optional error message

Returns an SyntaxError instance

Return type SyntaxError

exception kong.lexer.**SyntaxError**(*offset*, *message=None*)
An exception that rises when the syntax is invalid.

offset = None
(numbers.Integral) The errored offset of the string.

get_line(*string*)
Gets the errored line number from the code string.

Parameters **string** (basestring) – code string

Returns 0-based line number

Return type numbers.Integral

get_column(*string*)
Gets the errored column number of the line from the code string.

Parameters **string** (basestring) – code string

Returns 0-based column number

Return type numbers.Integral

1.4 kong.version — Version data

kong.version.**VERSION_INFO = (0, 1, 0)**
(tuple) The 3-tuple contains the version numbers e.g. (0, 1, 2).

kong.version.**VERSION = '0.1.0'**
(basestring) The version string e.g. '0.1.2'.

CHAPTER 2

Indices and tables

- *genindex*
- *modindex*
- *search*

k