SMPTE VC-2 Pseudocode Parsing Software

Release 1.0.0

BBC

Mar 05, 2021

CONTENTS

CHAPTER

ONE

INTRODUCTION

This manual describes the VC-2 pseudocode parsing software. This software provides facilities for parsing, translating and formatting the pseudocode language described in SMPTE ST 2042-1 (VC-2) series of standards documents.

In *Pseudocode Parser* (page 3) the *vc2_pseudocode_parser.parser* (page 3) module is introduced which implements a parser and Abstract Syntax Tree (AST) for the pseudocode language. This forms the basis of the other tools provided by this software and also may be used directly if desired.

In *Pseudocode to Python translation* (page 15), the `vc2-pseudocode-to-python` command (and associated *vc2_pseudocode_parser.python_transformer* (page 15) Python module) are introduced. These produce automatic translations of VC-2 pseudocode listings into valid Python.

In *Pseudocode to SMPTE Word Document translation* (page 17), the `vc2-pseudocode-to-docx` command (and associated *vc2_pseudocode_parser.docx_transformer* (page 17) Python module) are introduced. These generate Word (docx) documents containing pretty-printed and syntax highlighted versions of a VC-2 pseudocode listing. Suplimentrary to this, *Word Document Construction* (page 19) gives additional details of the Word document generation process.

Finally, you can find the source code for `vc2_pseudocode_parser` on GitHub[1].

Note: This documentation is also available to browse online in HTML format[2].

[1] https://github.com/bbc/vc2_pseudocode_parser/
[2] https://bbc.github.io/vc2_pseudocode_parser/

CHAPTER

TWO

PSEUDOCODE PARSER

The *vc2_pseudocode_parser.parser* (page 3) module contains a parser and associated Abstract Syntax Tree (AST) representation for the pseudocode language used within the VC-2 specifications [VC2].

A quick-start example illustrating much of the pseudocode syntax and basic usage of this module is given below:

```
>>> from vc2_pseudocode_parser.parser import parse

>>> source = '''
...     some_function(arg1, arg2, arg3):
...         # Assignments
...         hex_foo = 0xF00
...         sum = arg1 + arg2 + arg3
...
...         # If statements
...         if (sum > 0):
...             sign = 1
...         else if (sum < 0):
...             sign = -1
...         else:
...             sign = 0
...
...         # For-each: loop over fixed set of values
...         sum2 = 0
...         for each value in arg1, arg2, arg3:
...             sum2 += value
...
...         # For: loop over range of integers
...         sum_1_to_100 = 0
...         for n = 1 to 100:
...             sum_1_to_100 += n
...
...         # While loop
...         count = 0
...         while (sum > 0):
...             sum //= 2
...             count += 1
...
...         # Maps (like Python's dictionaries
...         map = {}
...
...         # Maps subscripted with labels
...         map[foo] = 123
...         map[bar] = 321
...
...         # Labels are first-class values (and are defined by their first use)
...         label = baz
...         map[label] = 999
...
...         # Function calls
```

(continues on next page)

(continued from previous page)

```
...         foo(map)
...
...         # Return from functions
...         return count
... '''

>>> ast = parse(source)

>>> ast.functions[0].name
'some_function'

>>> assignment = ast.functions[0].body[0]
>>> assignment.variable
Variable(offset=68, name='hex_foo')
>>> assignment.value
NumberExpr(offset=78, offset_end=83, value=3840, display_base=16, display_digits=3)
```

2.1 Parser

A pseudocode snippet may be parsed into an Abstract Syntax Tree (AST) using the *parse()* (page 4) function:

parse(*string*)

Parse a pseudocode listing into an abstract syntax tree (*Listing* (page 5)).

May raise a *ParseError* (page 4) or *ASTConstructionError* (page 4) exception on failure.

Parsing failures will result of one of the exceptions below being raised. In all cases, the `str`[3] representation of these errors produces a user-friendly description of the problem.

For example:

```
>>> from vc2_pseudocode_parser.parser import ParseError, ASTConstructionError

>>> try:
...     ast = parse("foo(): return (a + 3")
... except (ParseError, ASTConstructionError) as e:
...     print(str(e))
At line 1 column 21:
    foo(): return (a + 3
                        ^
Expected ')' or <operator>
```

exception ParseError

Re-exported from `peggie.ParseError`[4].

exception ASTConstructionError(*line*, *column*, *snippet*)

Exceptions thrown during construction of an AST.

exception LabelUsedAsVariableNameError(*line*, *column*, *snippet*, *variable_name*)

Bases: `vc2_pseudocode_parser.parser.ast.ASTConstructionError`

Thrown when a name previously used as a label is assigned to like a variable.

exception CannotSubscriptLabelError(*line*, *column*, *snippet*, *label_name*)

Bases: `vc2_pseudocode_parser.parser.ast.ASTConstructionError`

Thrown when name which has previously used as a label is subscripted like a variable.

[3] https://docs.python.org/3/library/stdtypes.html#str
[4] https://peggie.readthedocs.io/en/latest/index.html#peggie.ParseError

2.2 Abstract Syntax Tree (AST)

The parser generates a fairly detailed AST containing both semantic and some non-semantic features of the source such as explicit uses of parentheses, comments and vertical whitespace. Nodes in the AST also include character indices of the corresponding source code enabling the construction of helpful error messages.

Every node in the AST is a subclass of the *ASTNode* (page 5) base class:

class ASTNode(*offset: int*[5], *offset_end: int*[6])

offset: **int**[7]
Index of first character in the source related to this node.

offset_end: **int**[8]
Index of the character after the final character related to this node.

2.2.1 AST Root (`Listing`)

The root element of a complete AST is *Listing* (page 5):

class Listing(*functions*, *leading_empty_lines=<factory>*)
The root of a pseudocode AST.

functions: **List[*vc2_pseudocode_parser.parser.ast.Function* (page 5)]**
List of *Function* (page 5) trees for each function defined in the tree.

leading_empty_lines: **List[*vc2_pseudocode_parser.parser.ast.EmptyLine* (page 12)]**
List of *EmptyLine* (page 12) trees relating to empty (or comment-only) lines at the start of the source listing.

This in turn is principally made up of a list of *Function* (page 5) nodes:

class Function(*offset*, *name*, *arguments*, *body*, *eol=None*)
A function definition:

```
<name>(<arguments[0]>, <arguments[1]>, <...>): <eol>
    <body>
```

name: **str**[9]
The name of the function.

arguments: **List[*vc2_pseudocode_parser.parser.ast.Variable* (page 11)]**
The *Variable* (page 11) objects corresponding with the arguments to this function.

body: **List[*vc2_pseudocode_parser.parser.ast.Stmt* (page 6)]**
The list of *Stmt* (page 6) which make up this function.

eol: **Optional[*vc2_pseudocode_parser.parser.ast.EOL* (page 12)] = None**
EOL (page 12) at the end of the function heading.

[5] https://docs.python.org/3/library/functions.html#int
[6] https://docs.python.org/3/library/functions.html#int
[7] https://docs.python.org/3/library/functions.html#int
[8] https://docs.python.org/3/library/functions.html#int
[9] https://docs.python.org/3/library/stdtypes.html#str

2.2.2 Statements

AST nodes representing statements in the AST are subclasses of *Stmt* (page 6).

class Stmt(*offset*, *offset_end*)
Base-class for all statement AST nodes.

If-else-if-else statements are defined as follows:

class IfElseStmt(*if_branches*, *else_branch=None*)
An `if` statement with an arbitrary number of `else if` clauses and optional `else` clause.

A `if` statement is broken as illustrated by the following example:

```
if (condition0):        # \ if_branches[0]
    body0               # /
else if (condition1):   # \ if_branches[1]
    body1               # /
else if (condition2):   # \ if_branches[2]
    body2               # /
else:                   # \ else_branch
    body3               # /
```

if_branches: List[*vc2_pseudocode_parser.parser.ast.IfBranch* (page 6)]
The opening `if` clause followed by any `else if` clauses are represented as a series of *IfBranch* (page 6).

else_branch: Optional[*vc2_pseudocode_parser.parser.ast.ElseBranch* (page 6)] = None
If an `else` clause is present, a *ElseBranch* (page 6) giving its contents.

class IfBranch(*offset*, *condition*, *body*, *eol=None*)
An `if` or `else if` clause in an if-else-if-else statement:

```
if (<condition>): <eol>
    <body>
```

Or:

```
else if (<condition>): <eol>
    <body>
```

condition: *vc2_pseudocode_parser.parser.ast.Expr* (page 9)
The *Expr* (page 9) representing the condition for the branch condition.

body: List[*vc2_pseudocode_parser.parser.ast.Stmt* (page 6)]
The list of *Stmt* (page 6) to execute when the condition is True.

eol: Optional[*vc2_pseudocode_parser.parser.ast.EOL* (page 12)] = None
EOL (page 12) following the `if` or `else if` clause heading.

class ElseBranch(*offset*, *body*, *eol=None*)
An `else` clause in an if-else-if-else statement:

```
else: <eol>
    <body>
```

body: List[*vc2_pseudocode_parser.parser.ast.Stmt* (page 6)]
The list of *Stmt* (page 6) to execute when the else branch is reached.

eol: Optional[*vc2_pseudocode_parser.parser.ast.EOL* (page 12)] = None
EOL (page 12) following the `else` clause heading.

For-each loops are defined as follows:

class ForEachStmt(*offset*, *variable*, *values*, *body*, *eol=None*)
A `for each` loop:

```
for each <variable> in <values[0]>, <values[1]>, <...>: <eol>
    <body>
```

variable: *vc2_pseudocode_parser.parser.ast.Variable* **(page 11)**
The loop *Variable* (page 11).

values: **List[***vc2_pseudocode_parser.parser.ast.Expr* **(page 9)]**
The *Expr* (page 9) giving the set of values the loop will iterate over.

body: **List[***vc2_pseudocode_parser.parser.ast.Stmt* **(page 6)]**
The list of *Stmt* (page 6) to execute in each iteration.

eol: **Optional[***vc2_pseudocode_parser.parser.ast.EOL* **(page 12)] = None**
EOL (page 12) following the `for each` heading.

For loops are defined as follows:

class ForStmt(*offset*, *variable*, *start*, *end*, *body*, *eol=None*)
A `for` loop:

```
for <variable> = <start> to <end>: <eol>
    <body>
```

variable: *vc2_pseudocode_parser.parser.ast.Variable* **(page 11)**
The loop *Variable* (page 11).

start: *vc2_pseudocode_parser.parser.ast.Expr* **(page 9)**
The *Expr* (page 9) giving the loop starting value.

end: *vc2_pseudocode_parser.parser.ast.Expr* **(page 9)**
The *Expr* (page 9) giving the (inclusive) loop ending value.

body: **List[***vc2_pseudocode_parser.parser.ast.Stmt* **(page 6)]**
The list of *Stmt* (page 6) to execute in each iteration.

eol: **Optional[***vc2_pseudocode_parser.parser.ast.EOL* **(page 12)] = None**
EOL (page 12) following the `for` heading.

While loops are defined as follows:

class WhileStmt(*offset*, *condition*, *body*, *eol=None*)
A `while` loop:

```
while (<condition>): <eol>
    <body>
```

condition: *vc2_pseudocode_parser.parser.ast.Expr* **(page 9)**
The *Expr* (page 9) representing the loop condition.

body: **List[***vc2_pseudocode_parser.parser.ast.Stmt* **(page 6)]**
The list of *Stmt* (page 6) to execute in each iteration.

eol: **Optional[***vc2_pseudocode_parser.parser.ast.EOL* **(page 12)] = None**
EOL (page 12) following the `while` heading.

Function call statements are defined as follows:

class FunctionCallStmt(*call*, *eol*)
A statement which represents a call to a function.

call: *vc2_pseudocode_parser.parser.ast.FunctionCallExpr* **(page 10)**
The *FunctionCallExpr* (page 10) which defines the function call.

eol: *vc2_pseudocode_parser.parser.ast.EOL* **(page 12)**
EOL (page 12) following the function call.

Return statements are defined as follows:

class ReturnStmt(*offset*, *value*, *eol*)

A return statement:

```
return <value> <eol>
```

value: ***vc2_pseudocode_parser.parser.ast.Expr*** **(page 9)**

A *Expr* (page 9) giving the value to be returned.

eol: ***vc2_pseudocode_parser.parser.ast.EOL*** **(page 12)**

EOL (page 12) following the return statement.

Assignment statements are defined as follows:

class AssignmentStmt(*variable*, *op*, *value*, *eol*)

A simple or compound assignment statement:

```
<variable> <op> <value> <eol>  # e.g. x += 1 + 2
```

variable: **Union[*vc2_pseudocode_parser.parser.ast.Variable* (page 11), *vc2_pseudocode*

The *Variable* (page 11) or *Subscript* (page 11) being assigned to.

op: ***vc2_pseudocode_parser.parser.operators.AssignmentOp*** **(page 8)**

The type of assignment being performed (`AssignmentOp`).

value: ***vc2_pseudocode_parser.parser.ast.Expr*** **(page 9)**

The *Expr* (page 9) giving the value being assigned.

eol: ***vc2_pseudocode_parser.parser.ast.EOL*** **(page 12)**

EOL (page 12) following the assignment statement.

The following assignment operators are defined:

class AssignmentOp(*value*)

An enumeration.

assign = '='

add_assign = '+='

sub_assign = '-='

mul_assign = '*='

idiv_assign = '//='

pow_assign = '='**

and_assign = '&='

xor_assign = '^='

or_assign = '|='

lsh_assign = '<<='

rsh_assign = '>>='

2.2.3 Expressions

AST nodes representing expressions in the AST are subclasses of *Expr* (page 9).

class Expr(*offset: int*[10], *offset_end: int*[11])

Unary expressions are defined as follows:

class UnaryExpr(*offset*, *op*, *value*)
A unary expression, e.g. `-foo`.

op: ***vc2_pseudocode_parser.parser.operators.UnaryOp*** **(page 9)**
The operator (*UnaryOp* (page 9))

value: ***vc2_pseudocode_parser.parser.ast.Expr*** **(page 9)**
The *Expr* (page 9) the operator applies to.

With unary operators enumerated as:

class UnaryOp(*value*)
An enumeration.

plus = '+'

minus = '-'

bitwise_not = '~'

logical_not = 'not'

Binary expressions are defined as follows:

class BinaryExpr(*lhs*, *op*, *rhs*)
A binary expression, e.g. `a + 1`.

lhs: ***vc2_pseudocode_parser.parser.ast.Expr*** **(page 9)**
The *Expr* (page 9) on the left-hand side of the expression.

op: ***vc2_pseudocode_parser.parser.operators.BinaryOp*** **(page 9)**
The operator (*BinaryOp* (page 9))

rhs: ***vc2_pseudocode_parser.parser.ast.Expr*** **(page 9)**
The *Expr* (page 9) on the right-hand side of the expression.

With binary operators enumerated as:

class BinaryOp(*value*)
An enumeration.

logical_or = 'or'

logical_and = 'and'

eq = '=='

ne = '!='

lt = '<'

le = '<='

gt = '>'

ge = '>='

bitwise_or_ = '|'

bitwise_xor = '^'

bitwise_and_ = '&'

[10] https://docs.python.org/3/library/functions.html#int
[11] https://docs.python.org/3/library/functions.html#int

lsh = '<<'

rsh = '>>'

add = '+'

sub = '-'

mul = '*'

idiv = '//'

mod = '%'

pow = ''**

Calls to functions are defined as follows:

class FunctionCallExpr(*offset*, *offset_end*, *name*, *arguments*)
A call to a function:

```
<name>(<arguments[0]>, <arguments[1]>, <...>)  # e.g. foo(1, 2, 3)
```

name: **str**[12]
The name of the function to be called.

arguments: **List[*vc2_pseudocode_parser.parser.ast.Expr* (page 9)]**
The list of *Expr* (page 9) giving the arguments to the call.

Uses of variables and subscripted variables are defined as follows:

class VariableExpr(*variable*)
A use of a variable or subscripted variable.

variable: **Union[*vc2_pseudocode_parser.parser.ast.Variable* (page 11), *vc2_pseudocod***
The *Variable* (page 11) or *Subscript* (page 11) used.

Value literals are defined as follows:

class BooleanExpr(*offset*, *value*)
A boolean literal, i.e. `True` and `False`.

value: **bool**[13]
The boolean value.

class NumberExpr(*offset*, *offset_end*, *value*, *display_base=10*, *display_digits=1*)
A numerical literal integer, e.g. `123` or `0xF00`.

value: **int**[14]
The (parsed) integer value.

display_base: **int**[15] **= 10**
The base which the literal was encoded in.

display_digits: **int**[16] **= 1**
The number of digits used in the literal, including leading zeros but excluding any prefix (e.g. `0x` or `0b`).

class EmptyMapExpr(*offset*, *offset_end*)
An empty map literal (e.g. `{}`).

class LabelExpr(*label*)
A label literal.

[12] https://docs.python.org/3/library/stdtypes.html#str
[13] https://docs.python.org/3/library/functions.html#bool
[14] https://docs.python.org/3/library/functions.html#int
[15] https://docs.python.org/3/library/functions.html#int
[16] https://docs.python.org/3/library/functions.html#int

label: *vc2_pseudocode_parser.parser.ast.Label* **(page 11)**
The *Label* (page 11) used.

For parenthesised expressions, e.g. `(1 + 2)`, the presence of the parentheses is explicitly marked in the AST. While this is not semantically important (since evaluation order is explicit in an AST) it may be helpful in retaining parentheses added for human legibility when translating the pseudocode into other forms.

class ParenExpr(*offset*, *offset_end*, *value*)
A parenthesised expression.

value: *vc2_pseudocode_parser.parser.ast.Expr* **(page 9)**
The parenthesised *Expr* (page 9)

2.2.4 Variables and subscripts

A *Variable* (page 11) is defined as follows:

class Variable(*offset*, *name*)
A use of a variable.

name: **str**[17]
The name of the variable.

Variables may be subscripted (multiple times) and this is represented by a nesting of *Subscript* (page 11) objects:

class Subscript(*offset_end*, *variable*, *subscript*)
A subscripted variable (e.g. `x[1]`).

variable: **Union[***vc2_pseudocode_parser.parser.ast.Subscript* **(page 11),** *vc2_pseudoco*
The *Variable* (page 11) being subscripted (or *Subscript* (page 11) in the case of a multiply subscripted variable.

subscript: *vc2_pseudocode_parser.parser.ast.Expr* **(page 9)**
The *Expr* (page 9) giving the subscript value.

2.2.5 Labels

Labels are defined as follows:

class Label(*offset*, *name*)
A label value.

name: **str**[18]
The label name.

Note: Labels and variables are syntactically ambiguous in the pseudocode language but are disambiguated in the AST. Names in the pseudocode are deemed to be variables if they correspond with function arguments, loop variables or assignment targets within a function's namespace. All other names are deemed to be labels.

---

[17] https://docs.python.org/3/library/stdtypes.html#str
[18] https://docs.python.org/3/library/stdtypes.html#str

2.2.6 Comments and vertical whitespace

All comments and vertical whitespace (i.e. blank lines) are captured by the AST. This enables these non-semantic components to be retained in language translations.

class Comment(*offset*, *string*)
An end-of-line comment.

string: str[19]
The comment string, including leading '#' but not trailing newline.

class EmptyLine(*offset*, *offset_end*, *comment=None*)
Represents an empty (or comment-only) line in the source.

comment: Optional[*vc2_pseudocode_parser.parser.ast.Comment* (page 12)] = None
The *Comment* (page 12) on this line, if present.

Simple statements are syntactically terminated by an optional comment followed by a newline and then a number of empty or comment-only lines. These details are captured by the *EOL* (page 12) node.

class EOL(*offset*, *offset_end*, *comment=None*, *empty_lines=<factory>*)
The end-of-line terminator following a statement.

comment: Optional[*vc2_pseudocode_parser.parser.ast.Comment* (page 12)] = None
A *Comment* (page 12) on the same line as the statement, if present.

empty_lines: List[*vc2_pseudocode_parser.parser.ast.EmptyLine* (page 12)]
Any trailing *EmptyLine* (page 12) after this statement.

For example, given a snippet as follows:

```
foo() # Comment 0

bar() # Comment 1
# Comment 2

# Comment 3

baz()  # Comment 4
```

Here the function call `bar()` would be captured by a *FunctionCallStmt* (page 7). The *FunctionCallStmt.eol* (page 7) would contain an *EOL* (page 12) with *EOL.comment* (page 12) set to a *Comment* (page 12) containing `# Comment 1` and *EOL.empty_lines* (page 12) the following four lines (and their comments).

Function declarations, `if`, `else if`, `else`, `for each`, `for` and `while` headings are separated from their bodies by a `:` and optional *EOL* (page 12). When the `:` is followed by a newline, an *EOL* (page 12) object will be given, but for in-line definitions, the *EOL* (page 12) will be omitted. For example:

```
func1():
    foo()

func2(): foo()
```

Here the *Function* (page 5) for `func1` will have *Function.eol* (page 5) set to an *EOL* (page 12) node for `func2` it will be `None`.

[19] https://docs.python.org/3/library/stdtypes.html#str

2.3 Grammar

The pseudocode language grammar can be described by its peggie[20] grammar:

```
# Grammar for the VC-2 specification pseudocode language
start <- any_ws @=function+ eof

# Function definition
function           <- identifier ws function_arguments ws stmt_block
function_arguments <- "(" ws (identifier ws ("," ws identifier ws)* ","?)?  ws ")"

# A series of statements
stmt_block <- ":" ws single_line_stmt
            / ":" eol @>((@=stmt)+)

# Statements (all end with an eol)
stmt <- if_else_stmt
      / for_each_stmt
      / for_stmt
      / while_stmt
      / function_call_stmt
      / return_stmt
      / assignment_stmt

single_line_stmt <- function_call_stmt
                  / return_stmt
                  / assignment_stmt

function_call_stmt <- function_call eol
if_else_stmt       <- @=("if" ws condition ws stmt_block)
                      @=(@=("else" ws_ "if" ws condition ws stmt_block)*)
                      @=(("else" ws stmt_block)?)
for_each_stmt      <- "for" ws_ "each" ws_ identifier ws_ "in" ws_ for_each_list␣
↪ws stmt_block
for_stmt           <- "for" ws_ identifier ws "=" ws expr ws_ "to" ws_ expr ws␣
↪stmt_block
while_stmt         <- "while" ws condition ws stmt_block
assignment_stmt    <- variable ws assignment_op ws expr eol
return_stmt        <- "return" ws_ expr eol

condition      <- "(" ws expr ws ")"
for_each_list  <- expr (ws "," ws expr)*
assignment_op  <- r"(\+|-|\*|//|\*\*|&|\^|\||<<|>>)?="

# Expressions (defined below in ascending order of precidence)
expr <- maybe_log_or_expr
maybe_log_or_expr  <- maybe_log_and_expr (ws_ "or"              ws_ maybe_log_and_
↪expr)*
maybe_log_and_expr <- maybe_log_not_expr (ws_ "and"             ws_ maybe_log_not_
↪expr)*
maybe_log_not_expr <- "not" ws_ maybe_log_not_expr / maybe_cmp_expr
maybe_cmp_expr     <- maybe_or_expr      (ws r"==|!=|<=|>=|<|>" ws maybe_or_expr)*
maybe_or_expr      <- maybe_xor_expr     (ws "|"                ws maybe_xor_expr)*
maybe_xor_expr     <- maybe_and_expr     (ws "^"                ws maybe_and_expr)*
maybe_and_expr     <- maybe_shift_expr   (ws "&"                ws maybe_shift_
↪expr)*
maybe_shift_expr   <- maybe_arith_expr   (ws r"<<|>>"           ws maybe_arith_
↪expr)*
maybe_arith_expr   <- maybe_prod_expr    (ws r"\+|-"            ws maybe_prod_
↪expr)*
```

(continues on next page)

[20] https://peggie.readthedocs.io/en/latest/index.html#module-peggie

(continued from previous page)

```
maybe_prod_expr    <- maybe_unary_expr   (ws r"\*|//|%"         ws maybe_unary_
↪expr)*
maybe_unary_expr   <- r"\+|-|~" ws maybe_unary_expr / maybe_pow_expr
maybe_pow_expr     <- maybe_paren_expr   (ws "**"               ws maybe_unary_
↪expr)*
maybe_paren_expr   <- "(" ws expr ws ")" / atom

# Atoms
atom <- function_call
      / variable
      / empty_map
      / boolean
      / number

variable  <- identifier (ws subscript)*
subscript <- "[" ws expr ws "]"

function_call           <- identifier ws function_call_arguments
function_call_arguments <- "(" ws (expr ws ("," ws expr ws)* ","?)? ws ")"

# Literals
identifier    <- !reserved_word r"[a-zA-Z_][a-zA-Z0-9_]*"
reserved_word <- r
↪"(if|else|elif|elseif|for|each|foreach|in|to|while|return|[Tt]rue|[Ff]alse|and|or|not)(?
↪![a-zA-Z0-9_])"
boolean       <- ("True" / "False")
number        <- r"(0[bB][01]+)|(0[xX][0-9a-fA-F]+)|([0-9]+)"
empty_map     <- "{" ws "}"

# Whitespace and comments
comment  <- r"#((?![\n\r]).)*(\n|\r\n|\r|(?!.))"
ws       <- h_space?
ws_      <- h_space
any_ws   <- (comment / h_space / v_space)*
eol      <- h_space? (comment / v_space / eof) any_ws
h_space  <- r"[ \t]+"
v_space  <- "\n" / "\r\n" / "\r"
eof      <- !.
```

2.4 Operator precedence and associativity tables

A table of operator precedence and associativities is also provided which may be useful for, for example, producing pretty-printed outputs (with excess parentheses removed).

OPERATOR_PRECEDENCE_TABLE = {operator: int, ...}

BinaryOp (page 9) and *UnaryOp* (page 9) operator precedence scores. Higher scores mean higher precedence.

OPERATOR_ASSOCIATIVITY_TABLE = {operator: associativity, ...}

BinaryOp (page 9) and *UnaryOp* (page 9) operator associativities.

class Associativity(*value*)

Operator associativity types.

left = 'left'

right = 'right'

CHAPTER

THREE

PSEUDOCODE TO PYTHON TRANSLATION

The *vc2_pseudocode_parser.python_transformer* (page 15) module and `vc2-pseudocode-to-python` command line tool automatically translate pseudocode listings into valid Python.

In general, the translation between pseudocode and Python is 'obvious'. The only non-obvious part, perhaps, is that labels are translated into Python string literals. The output is pretty-printed in a style similar to the Black[21] code style with comments and vertical whitespace retained (in a semi-normalised fashion).

For example, the following pseudocode:

```
add(a, b, c):
    # A function which adds three numbers together
    total = 0  # An accumulator
    for each n in a, b, c:
        total += n
    return total

update_state(state):
    state[count] += 1
```

Is translated in the the following Python:

```
def add(a, b, c):
    """
    A function which adds three numbers together
    """
    total = 0  # An accumulator
    for n in [a, b, c]:
        total += n
    return total


def update_state(state):
    state["count"] += 1
```

3.1 Command-line utility

The `vc2-pseudocode-to-python` command line utility is provided which can convert a pseudocode listing into Python.

Example usage:

```
$ vc2-pseudocode-to-python input.pc output.py
```

[21] https://github.com/psf/black

3.2 Python API

The *pseudocode_to_python()* (page 16) utility function may be used to directly translate pseudocode into Python.

pseudocode_to_python(*pseudocode_source*, *indent=' '*, *generate_docstrings=True*, *add_translation_note=False*)

Transform a pseudocode listing into Python.

Will throw a *ParseError* (page 4) or *ASTConstructionError* (page 4) if the supplied pseudocode contains syntactic errors.

Parameters

pseudocode_source [str] The pseudocode source code to translate.

indent [str] The string to use for indentation in the generated Python source. Defaults to four spaces.

generate_docstrings [bool] If True, the first block of comments in the file and each function will be converted into a docstring. Otherwise they'll be left as ordinary comments. Defaulse to True.

add_translation_note [bool] If True, adds a comment to the top of the generated output indicating that this file was automatically translated from the pseudocode. Defaulse to False.

Example usage:

```
>>> from vc2_pseudocode_parser.python_transformer import pseudocode_to_python

>>> print(pseudocode_to_python("""
...     foo(state, a):
...         state[bar] = a + 1
... """))
def foo(state, a):
    state["bar"] = a + 1
```

CHAPTER

FOUR

PSEUDOCODE TO SMPTE WORD DOCUMENT TRANSLATION

The *vc2_pseudocode_parser.docx_transformer* (page 17) module and `vc2-pseudocode-to-docx` command line tool automatically translate pseudocode listings into syntax-highlighted SMPTE-style listings tables in a Word document.

As an example the following input:

```
padding_data(state):                                # Ref
    # Read a padding data block
    for i = 1 to state[next_parse_offset]-13:       # 10.5
        # NB: data is just discarded
        read_byte()                                 # A.2.2
```

Is transformed into the following output:

padding_data`(state):`	Ref
`  for i = 1 to state[`*`next_parse_offset`*`] - 13:`	10.5
`    read_byte()`	A.2.2

Note that:

- **Syntax highlighting has been applied**
  - Keywords are in bold (e.g. `for` and `to`)
  - Labels are italicised (e.g. `next_parse_offset`)
  - Variables and other values are in normal print
- Spacing is normalised (e.g. around the − operator)
- End-of-line comments are shown in a right-hand column
- Comments appearing on their own are omitted

4.1 Dependencies

To generate word documents the python-docx[22] library is used. This is an optional dependency of the `vc2_pseudocode_parser` software and must be installed separately, e.g. using:

```
$ pip install python-docx
```

[22] https://python-docx.readthedocs.io/en/latest/

4.2 Command-line utility

The `vc2-pseudocode-to-docx` command line utility is provided which can convert a pseudocode listing into a Word document.

Example usage:

```
$ vc2-pseudocode-to-docx input.pc output.docx
```

4.3 Python API

The *pseudocode_to_docx()* (page 18) utility function may be used to directly translate pseudocode into a Word document.

pseudocode_to_docx(*pseudocode_source*, *filename*)

Transform a pseudocode listing into a Word (docx) document.

Will throw a *ParseError* (page 4) *ASTConstructionError* (page 4) if the supplied pseudocode contains errors.

Example usage:

```
>>> from vc2_pseudocode_parser.docx_transformer import pseudocode_to_docx

>>> pseudocode_source = '''
...     foo(state, a):
...         state[bar] = a + 1
... '''
>>> pseudocode_to_docx(pseudocode_source, "/path/to/output.docx")
```

CHAPTER

FIVE

WORD DOCUMENT CONSTRUCTION

The *vc2_pseudocode_parser.docx_generator* (page 19) module implements a simplified wrapper around `docx` for generating Word documents containing SMPTE specification style code listings.

5.1 Document model

Documents are defined using a hierarchy of the following classes:

class ListingDocument(*body=<factory>*)

A document containing code listings. The root of the document heirarchy.

**body: List[Union[*vc2_pseudocode_parser.docx_generator.Paragraph* (page 19), *vc2_pse*

The contents of this document.

make_docx_document()

Construct a `docx.Document` object from this document (ready for saving as a file or further manipulation).

class Paragraph(*runs=[]*)

A paragraph of text consisting of a series of concatenated *Runs* (page 19) of text.

As a convenience, the constructor accepts either a list of *Runs* (page 19), a single *Run* (page 19) or a `str`[23].

Paragraph (page 19) objects support the + operator which will concatenate the runs in a pair of *Paragraphs* (page 19), producing a new *Paragraph* (page 19). You can also add *Run* (page 19) and `str`[24] to *Paragraphs* (page 19) with similar effect.

runs: List[*vc2_pseudocode_parser.docx_generator.Run* (page 19)]

The runs of text contained in this paragraph.

class Run(*text=''*, *style=None*)

A run of text within a paragraph, which will be rendered with a particular style (defined in *RunStyle* (page 19)).

text: str[25] **= ''**

style: Optional[*vc2_pseudocode_parser.docx_generator.RunStyle* (page 19)] = None

add_to_docx_paragraph(*docx_document*, *docx_paragraph*)

class RunStyle(*value*)

Text styles for runs.

pseudocode = 'Pseudocode'

pseudocode_fdef = 'Pseudocode Function Defnition'

pseudocode_keyword = 'Pseudocode Keyword'

[23] https://docs.python.org/3/library/stdtypes.html#str
[24] https://docs.python.org/3/library/stdtypes.html#str
[25] https://docs.python.org/3/library/stdtypes.html#str

pseudocode_label = 'Pseudocode Label'

class ListingTable(*rows*)
A code listing table giving the source code for a single function definiton.

If any lines in the listing contain a comment, the resulting table will have two colums with the code on the left and comments on the right. If no lines contain a comment, the table will have a single column.

rows: List[*vc2_pseudocode_parser.docx_generator.ListingLine* (page 20)]
The rows in the table.

class ListingLine(*code=<factory>*, *comment=<factory>*)
A single row in a *ListingTable* (page 20).

code: *vc2_pseudocode_parser.docx_generator.Paragraph* (page 19)

comment: *vc2_pseudocode_parser.docx_generator.Paragraph* (page 19)

BIBLIOGRAPHY

[VC2] SMPTE ST-2042-1 (VC-2)

INDEX

M

N

O

P

R

S

T

U

V

W

X