Compiler MATLAB to Python

Release 0.1.0

Artem Eroshev

Apr 30, 2021

Chapter 1

cmp

1.1 CMP package

1.1.1 Subpackages

Ast package

Submodules

Additive module

class `cmp.ast.additive.`**`MinusNode`** (*lhs, rhs*)

Bases: `cmp.ast.lhs_rhs_node.LhsRhsNode`

Node of minus operation

`lhs`

`rhs`

class `cmp.ast.additive.`**`PlusNode`** (*lhs, rhs*)

Bases: `cmp.ast.lhs_rhs_node.LhsRhsNode`

Node of plus operation

`lhs`

`rhs`

Array module

class `cmp.ast.array.`**`ArrayNode`** (*ident, content*)

Bases: `cmp.ast.node.Node`

Create array by defined rules Expected expression like >>> zeros(1, 5) and etc

`content`

`ident`

class `cmp.ast.array.`**`ArrayVectorNode`** (*content*)

Bases: `cmp.ast.node.Node`

Node of vector array Expected expression like >>> [1, 2, 3, 4 * 3] and etc

`content`

Assigment module

class `cmp.ast.assigment.`**`AssignmentNode`** (*lhs*, *rhs*)

Bases: `cmp.ast.lhs_rhs_node.LhsRhsNode`

Assignment object node: Lhs Rhs object

`lhs`

`rhs`

Comment module

class `cmp.ast.comment.`**`CommentNode`** (*comment*)

Bases: `cmp.ast.node.Node`

`comment`

Conditional statement module

class `cmp.ast.conditional_statement.`**`ConditionalNode`** (*main_stmt*)

Bases: `cmp.ast.node.Node`

Base node for conditional statements

`main_stmt`

class `cmp.ast.conditional_statement.`**`ElseIfClauseNode`** (*main_stmt*, *stmt_list*)

Bases: `cmp.ast.conditional_statement.ConditionalNode`

Node of elseif clause

`main_stmt`

`stmt_list`

class `cmp.ast.conditional_statement.`**`ManyBranchConditionalNode`** (*main_stmt*, *main_branch*, *alt_chain*, *alt_branch*)

Bases: `cmp.ast.conditional_statement.ConditionalNode`

Node of many conditional statement

`alt_branch`

`alt_chain`

`main_branch`

`main_stmt`

class `cmp.ast.conditional_statement.`**`SimpleConditionalNode`** (*main_stmt*, *stmt_list*)

Bases: `cmp.ast.conditional_statement.ConditionalNode`

Conditional node for one way statement >>> if (expression is True) >>>   do_something

`main_stmt`

`stmt_list`

class `cmp.ast.conditional_statement.`**`TwoBranchConditionalNode`** (*main_stmt*, *main_branch*, *alt_branch*)

Bases: `cmp.ast.conditional_statement.ConditionalNode`

Conditional node for two way statement >>> if (expression is True) >>>   do_something >>> else >>>   to_do

`alt_branch`

main_branch

main_stmt

Define clear module

class `cmp.ast.define_clear.`**`ClearNode`** (*id_list*)
Bases: `cmp.ast.node.Node`
Object of clear key word

id_list

Define global module

class `cmp.ast.define_global.`**`GlobalNode`** (*id_list*)
Bases: `cmp.ast.node.Node`
Object of global key word

id_list

Equality module

class `cmp.ast.equality.`**`NegativeEqualityNode`** (*lhs, rhs*)
Bases: `cmp.ast.lhs_rhs_node.LhsRhsNode`
Negative equality object node

lhs

rhs

class `cmp.ast.equality.`**`PositiveEqualityNode`** (*lhs, rhs*)
Bases: `cmp.ast.lhs_rhs_node.LhsRhsNode`
Positive equality object node

lhs

rhs

Error module

class `cmp.ast.error.`**`ErrorNode`** (*message*)
Bases: `cmp.ast.node.Node`
Node of error parsing

message

Finite unit module

class `cmp.ast.finite_unit.`**`ConstantNode`** (*const*)
Bases: `cmp.ast.node.Node`
Node of constant

const

class `cmp.ast.finite_unit.`**`IdentifierNode`** (*ident*)
Bases: `cmp.ast.node.Node`
Node of identifier

ident

class `cmp.ast.finite_unit.`**`SimpleNode`** (*content*)
Bases: `cmp.ast.node.Node`
Finite point in traverse tree

content

Function module

class `cmp.ast.function.`**`FunctionDeclareNode`** (*return_list, name*)
Bases: `cmp.ast.node.Node`
Declaration of function

name

return_list

class `cmp.ast.function.`**`FunctionNameNode`** (*name, input_list*)
Bases: `cmp.ast.node.Node`

input_list

name

class `cmp.ast.function.`**`FunctionNode`** (*declare, body*)
Bases: `cmp.ast.node.Node`
Node of function object

body

declare

Iterations module

class `cmp.ast.iterations.`**`ForLoopNode`** (*iterator, express, body*)
Bases: `cmp.ast.node.Node`
Object of FOR loop

body

express

iter

class `cmp.ast.iterations.`**`WhileLoopNode`** (*express, body*)
Bases: `cmp.ast.node.Node`

body

express

Jump statement module

class `cmp.ast.jump_stmt.`**`BreakNode`**
Bases: `cmp.ast.node.Node`
Key word BREAK

class `cmp.ast.jump_stmt.`**`ReturnNode`**
Bases: `cmp.ast.node.Node`
Key word RETURN

Lhs rhs node module

class `cmp.ast.lhs_rhs_node.`**`LhsRhsNode`** (*lhs, rhs*)
Bases: `cmp.ast.node.Node`
Base for lhs, rhs object node

lhs

rhs

Logic module

class `cmp.ast.logic.`**`AndNode`** (*lhs*, *rhs*)

Bases: `cmp.ast.lhs_rhs_node.LhsRhsNode`

Object of logic AND

lhs

rhs

class `cmp.ast.logic.`**`OrNode`** (*lhs*, *rhs*)

Bases: `cmp.ast.lhs_rhs_node.LhsRhsNode`

Object of logic OR

lhs

rhs

Multiplicative module

class `cmp.ast.multiplicative.`**`ArrayDivNode`** (*lhs*, *rhs*)

Bases: `cmp.ast.lhs_rhs_node.LhsRhsNode`

Object of divide array

lhs

rhs

class `cmp.ast.multiplicative.`**`ArrayMulNode`** (*lhs*, *rhs*)

Bases: `cmp.ast.lhs_rhs_node.LhsRhsNode`

Object of multiply array

lhs

rhs

class `cmp.ast.multiplicative.`**`ArrayPowerNode`** (*lhs*, *rhs*)

Bases: `cmp.ast.lhs_rhs_node.LhsRhsNode`

Object of power array

lhs

rhs

class `cmp.ast.multiplicative.`**`ArrayRDivNode`** (*lhs*, *rhs*)

Bases: `cmp.ast.lhs_rhs_node.LhsRhsNode`

Object of right divide array

lhs

rhs

class `cmp.ast.multiplicative.`**`DivideNode`** (*lhs*, *rhs*)

Bases: `cmp.ast.lhs_rhs_node.LhsRhsNode`

Object of divide

lhs

rhs

class `cmp.ast.multiplicative.`**`MultiplyNode`** (*lhs, rhs*)

Bases: `cmp.ast.lhs_rhs_node.LhsRhsNode`

Object of multiply

lhs

rhs

class `cmp.ast.multiplicative.`**`PowerNode`** (*lhs, rhs*)

Bases: `cmp.ast.lhs_rhs_node.LhsRhsNode`

Object of power

lhs

rhs

Node module

class `cmp.ast.node.`**`Node`**

Bases: `abc.ABC`

Base node for AST

Relational module

class `cmp.ast.relational.`**`GreaterEqualRelationalNode`** (*lhs, rhs*)

Bases: `cmp.ast.lhs_rhs_node.LhsRhsNode`

Greater or equal relational object node

lhs

rhs

class `cmp.ast.relational.`**`GreaterRelationalNode`** (*lhs, rhs*)

Bases: `cmp.ast.lhs_rhs_node.LhsRhsNode`

Greater relational object node

lhs

rhs

class `cmp.ast.relational.`**`LowerEqualRelationalNode`** (*lhs, rhs*)

Bases: `cmp.ast.lhs_rhs_node.LhsRhsNode`

Lower or equal object node

lhs

rhs

class `cmp.ast.relational.`**`LowerRelationalNode`** (*lhs, rhs*)

Bases: `cmp.ast.lhs_rhs_node.LhsRhsNode`

Lower object node

lhs

rhs

Root module

class `cmp.ast.root.`**`FileAST`** (*root*)

Bases: `cmp.ast.node.Node`

Entry point in AST

`root`

Sparse module

class `cmp.ast.sparse.`**`SparseNode`** (*lhs*, *rhs*)
Bases: `cmp.ast.lhs_rhs_node.LhsRhsNode`
Node of smudge value

`lhs`

`rhs`

Transpose module

class `cmp.ast.transpose.`**`TransposeNode`** (*expr*)
Bases: `cmp.ast.node.Node`
Node of transpose operation

`expr`

Unary expression module

class `cmp.ast.unary_expression.`**`UnaryExpressionNode`** (*unary_op*, *expr*)
Bases: `cmp.ast.node.Node`
Node of unary operator

`expr`

`unary_op`

Module contents

class `cmp.ast.`**`AndNode`** (*lhs*, *rhs*)
Bases: `cmp.ast.lhs_rhs_node.LhsRhsNode`
Object of logic AND

`lhs`

`rhs`

class `cmp.ast.`**`ArrayDivNode`** (*lhs*, *rhs*)
Bases: `cmp.ast.lhs_rhs_node.LhsRhsNode`
Object of divide array

`lhs`

`rhs`

class `cmp.ast.`**`ArrayMulNode`** (*lhs*, *rhs*)
Bases: `cmp.ast.lhs_rhs_node.LhsRhsNode`
Object of multiply array

`lhs`

`rhs`

class `cmp.ast.`**`ArrayNode`** (*ident*, *content*)
Bases: `cmp.ast.node.Node`
Create array by defined rules Expected expression like >>> zeros(1, 5) and etc

`content`

ident

class `cmp.ast.`**ArrayPowerNode** (*lhs*, *rhs*)

Bases: `cmp.ast.lhs_rhs_node.LhsRhsNode`

Object of power array

lhs

rhs

class `cmp.ast.`**ArrayRDivNode** (*lhs*, *rhs*)

Bases: `cmp.ast.lhs_rhs_node.LhsRhsNode`

Object of right divide array

lhs

rhs

class `cmp.ast.`**ArrayVectorNode** (*content*)

Bases: `cmp.ast.node.Node`

Node of vector array Expected expression like >>> [1, 2, 3, 4 * 3] and etc

content

class `cmp.ast.`**AssignmentNode** (*lhs*, *rhs*)

Bases: `cmp.ast.lhs_rhs_node.LhsRhsNode`

Assignment object node: Lhs Rhs object

lhs

rhs

class `cmp.ast.`**BreakNode**

Bases: `cmp.ast.node.Node`

Key word BREAK

class `cmp.ast.`**ClearNode** (*id_list*)

Bases: `cmp.ast.node.Node`

Object of clear key word

id_list

class `cmp.ast.`**CommentNode** (*comment*)

Bases: `cmp.ast.node.Node`

comment

class `cmp.ast.`**ConstantNode** (*const*)

Bases: `cmp.ast.node.Node`

Node of constant

const

class `cmp.ast.`**DivideNode** (*lhs*, *rhs*)

Bases: `cmp.ast.lhs_rhs_node.LhsRhsNode`

Object of divide

lhs

rhs

class `cmp.ast.`**`ElseIfClauseNode`** (*main_stmt*, *stmt_list*)
Bases: `cmp.ast.conditional_statement.ConditionalNode`
Node of elseif clause

main_stmt

stmt_list

class `cmp.ast.`**`ErrorNode`** (*message*)
Bases: `cmp.ast.node.Node`
Node of error parsing

message

class `cmp.ast.`**`FileAST`** (*root*)
Bases: `cmp.ast.node.Node`
Entry point in AST

root

class `cmp.ast.`**`ForLoopNode`** (*iterator*, *express*, *body*)
Bases: `cmp.ast.node.Node`
Object of FOR loop

body

express

iter

class `cmp.ast.`**`FunctionDeclareNode`** (*return_list*, *name*)
Bases: `cmp.ast.node.Node`
Declaration of function

name

return_list

class `cmp.ast.`**`FunctionNameNode`** (*name*, *input_list*)
Bases: `cmp.ast.node.Node`

input_list

name

class `cmp.ast.`**`FunctionNode`** (*declare*, *body*)
Bases: `cmp.ast.node.Node`
Node of function object

body

declare

class `cmp.ast.`**`GlobalNode`** (*id_list*)
Bases: `cmp.ast.node.Node`
Object of global key word

id_list

class `cmp.ast.`**`GreaterEqualRelationalNode`** (*lhs*, *rhs*)
Bases: `cmp.ast.lhs_rhs_node.LhsRhsNode`
Greater or equal relational object node

lhs

rhs

class `cmp.ast.`**`GreaterRelationalNode`** (*lhs*, *rhs*)
Bases: `cmp.ast.lhs_rhs_node.LhsRhsNode`
Greater relational object node

lhs

rhs

class `cmp.ast.`**`IdentifierNode`** (*ident*)
Bases: `cmp.ast.node.Node`
Node of identifier

ident

class `cmp.ast.`**`LowerEqualRelationalNode`** (*lhs*, *rhs*)
Bases: `cmp.ast.lhs_rhs_node.LhsRhsNode`
Lower or equal object node

lhs

rhs

class `cmp.ast.`**`LowerRelationalNode`** (*lhs*, *rhs*)
Bases: `cmp.ast.lhs_rhs_node.LhsRhsNode`
Lower object node

lhs

rhs

class `cmp.ast.`**`ManyBranchConditionalNode`** (*main_stmt*, *main_branch*, *alt_chain*, *alt_branch*)
Bases: `cmp.ast.conditional_statement.ConditionalNode`
Node of many conditional statement

alt_branch

alt_chain

main_branch

main_stmt

class `cmp.ast.`**`MinusNode`** (*lhs*, *rhs*)
Bases: `cmp.ast.lhs_rhs_node.LhsRhsNode`
Node of minus operation

lhs

rhs

class `cmp.ast.`**`MultiplyNode`** (*lhs*, *rhs*)

Bases: `cmp.ast.lhs_rhs_node.LhsRhsNode`

Object of multiply

`lhs`

`rhs`

class `cmp.ast.`**`NegativeEqualityNode`** (*lhs*, *rhs*)

Bases: `cmp.ast.lhs_rhs_node.LhsRhsNode`

Negative equality object node

`lhs`

`rhs`

class `cmp.ast.`**`Node`**

Bases: `abc.ABC`

Base node for AST

class `cmp.ast.`**`OrNode`** (*lhs*, *rhs*)

Bases: `cmp.ast.lhs_rhs_node.LhsRhsNode`

Object of logic OR

`lhs`

`rhs`

class `cmp.ast.`**`PlusNode`** (*lhs*, *rhs*)

Bases: `cmp.ast.lhs_rhs_node.LhsRhsNode`

Node of plus operation

`lhs`

`rhs`

class `cmp.ast.`**`PositiveEqualityNode`** (*lhs*, *rhs*)

Bases: `cmp.ast.lhs_rhs_node.LhsRhsNode`

Positive equality object node

`lhs`

`rhs`

class `cmp.ast.`**`PowerNode`** (*lhs*, *rhs*)

Bases: `cmp.ast.lhs_rhs_node.LhsRhsNode`

Object of power

`lhs`

`rhs`

class `cmp.ast.`**`ReturnNode`**

Bases: `cmp.ast.node.Node`

Key word RETURN

class `cmp.ast.`**`SimpleConditionalNode`** (*main_stmt*, *stmt_list*)

Bases: `cmp.ast.conditional_statement.ConditionalNode`

Conditional node for one way statement >>> if (expression is True) >>>   do_something

main_stmt

stmt_list

class `cmp.ast.`**SimpleNode** (*content*)

Bases: `cmp.ast.node.Node`

Finite point in traverse tree

content

class `cmp.ast.`**SparseNode** (*lhs, rhs*)

Bases: `cmp.ast.lhs_rhs_node.LhsRhsNode`

Node of smudge value

lhs

rhs

class `cmp.ast.`**TransposeNode** (*expr*)

Bases: `cmp.ast.node.Node`

Node of transpose operation

expr

class `cmp.ast.`**TwoBranchConditionalNode** (*main_stmt, main_branch, alt_branch*)

Bases: `cmp.ast.conditional_statement.ConditionalNode`

Conditional node for two way statement >>> if (expression is True) >>> do_something >>> else >>> to_do

alt_branch

main_branch

main_stmt

class `cmp.ast.`**UnaryExpressionNode** (*unary_op, expr*)

Bases: `cmp.ast.node.Node`

Node of unary operator

expr

unary_op

class `cmp.ast.`**WhileLoopNode** (*express, body*)

Bases: `cmp.ast.node.Node`

body

express

CLI package

Submodules

CLI module

class `cmp.cli.cli.`**Command** (**args, **kwargs*)

Bases: `argparse.ArgumentParser`, `cmp.helpers.log.LogMixin`, `cmp.helpers.singleton.Singleton`

CLI interface class

execute ( )
Entry point of program CMP for executed in command line

Return type `None`

Handlers check module

class `cmp.cli.handlers_check.`**AbstractHandler**
Bases: `cmp.cli.handlers_check.Handler`, `cmp.helpers.log.LogMixin`
Abstract handler for CLI interface CMP

abstract **handle** (*args*)

Return type `Optional[str]`

set_next (*handler*)

Return type `Handler`

class `cmp.cli.handlers_check.`**CheckOutputFile**
Bases: `cmp.cli.handlers_check.AbstractHandler`
Check -of/–output-file key. If it is true, result data will write in file by directed file. In default result show in output stream

handle (*args*)

Return type `Optional[str]`

class `cmp.cli.handlers_check.`**CheckServerKey**
Bases: `cmp.cli.handlers_check.AbstractHandler`
Check server option in input stream. If key -S/–server is true, TCP server will be started

handle (*args*)

Return type `Optional[str]`

static **network_execute** (*message*, *parser*)
Special entry point of program CMP for working in TCP server

Return type `Optional[str]`

class `cmp.cli.handlers_check.`**CheckStringKey**
Bases: `cmp.cli.handlers_check.AbstractHandler`
Check -s/–string and -p/–path key in input stream. Only one of two keys expected. If –string is true, data will get from input stream. If –path is true, data will read from file by directed path

handle (*args*)

Return type `Optional[str]`

class `cmp.cli.handlers_check.`**GetResult**
Bases: `cmp.cli.handlers_check.AbstractHandler`
After checks all keys this class generated result

handle (*args*)

Return type `Optional[str]`

class `cmp.cli.handlers_check.`**Handler**
Bases: `abc.ABC`
Base handler without subject area

abstract **handle** (*args*)

Return type `Optional[str]`

abstract **set_next** (*handler*)

Return type `Handler`

Module contents

Grammar package

Subpackages

Submodules

Lexer module

class `cmp.grammar.lexer.`**Lexer**

Bases: `cmp.helpers.log.LogMixin`

Executive lexer object. Containing primary tokens. This class give next token

D = '[0-9]'

E = '[DdEe][+-]?[0-9]+'

L = '[a-zA-Z_]'

constant = '[0-9]+([DdEe][+-]?[0-9]+)?|[0-9]*"."[0-9]+([DdEe][+-]?[0-9]+)?|[0-9]+"."[0-9]*([DdEe][+-]?[0-9]+)?'

constant_1 = '[0-9]+([DdEe][+-]?[0-9]+)?'

constant_2 = '[0-9]*"."[0-9]+([DdEe][+-]?[0-9]+)?'

constant_3 = '[0-9]+"."[0-9]*([DdEe][+-]?[0-9]+)?'

identifier = '[a-zA-Z_]([a-zA-Z_]|[0-9])*'

input (*data_*)

Return type `None`

keywords = {'break': 'BREAK', 'clear': 'CLEAR', 'else': 'ELSE', 'elseif': 'ELSEIF', 'end': 'END', 'for': 'FOR', 'function': 'FUNCTION', 'global': 'GLOBAL', 'if': 'IF', 'return': 'RETURN', 'while': 'WHILE'}

literals = ['~', ';', ',', ':', '=', '(', ')', '[', ']', '&', '-', '+', '*', '/', '\\', '>', '<', '|']

states = (('string', 'exclusive'),)

t_ANY_COMMENT (*token_*)

[%]|[n]

Return type `LexToken`

t_ARRAY_DIV = '\\./'

t_ARRAY_MUL = '\\.\\*'

t_ARRAY_POW = '\\.\\^'

t_ARRAY_RDIV = '\\.//'

t_CONSTANT (*token_*)

Return type `LexToken`

t_EQ_OP = '=='

t_GE_OP = '\\>='

t_IDENTIFIER (*token_*)

Return type LexToken

t_LE_OP = '<='

t_NEWLINE (*token_*)
n

Return type LexToken

t_NE_OP = '(~=)|(!=)'

t_STRING_LITERAL (*token_*)
'[^'n]*'

Return type LexToken

t_TRANSPOSE (*token_*)

Return type LexToken

t_error (*token_*)
Error handler lexer

Return type None

t_ignore_WHITESPACE = '\\s+'

t_string_TCOMMENT = '[^\\n]+'

t_string_error (*token_*)
Error handler lexer for string state

Return type None

t_string_ignore = ''

token ( )

Return type LexToken

tokens = ('IDENTIFIER', 'CONSTANT', 'STRING_LITERAL', 'LE_OP', 'GE_OP', 'EQ_OP', 'NE_OP', 'ARRAY_MUL', 'ARRAY_POW', 'ARRAY_DIV', 'ARRAY_RDIV', 'TRANSPOSE', 'NEWLINE', 'COMMENT', 'TCOMMENT', 'FOR', 'WHILE', 'BREAK', 'IF', 'ELSE', 'ELSEIF', 'END', 'FUNCTION', 'RETURN', 'GLOBAL', 'CLEAR')

transpose = "'|\\.'"

transpose_1 = "'"

transpose_2 = "\\.'"

Parser module

class cmp.grammar.parser.**Parser** (*lexer=<class 'cmp.grammar.lexer.Lexer'>, yacc_debug=False*)
Bases: object
Executive parser object. Containing primary reduce rules. This class build AST

errors ( )

Return type Iterator[str]

handlers = {'!=': <class 'cmp.ast.equality.NegativeEqualityNode'>, '&': <class 'cmp.ast.logic.AndNode'>, '*': <class 'cmp.ast.multiplicative.MultiplyNode'>, '+': <class 'cmp.ast.additive.PlusNode'>, '-': <class 'cmp.ast.additive.MinusNode'>, '.*': <class 'cmp.ast.multiplicative.ArrayMulNode'>, './': <class 'cmp.ast.multiplicative.ArrayDivNode'>, './/': <class 'cmp.ast.multiplicative.ArrayRDivNode'>, '.^': <class 'cmp.ast.multiplicative.ArrayPowerNode'>, '/': <class 'cmp.ast.multiplicative.DivideNode'>, '<': <class 'cmp.ast.relational.LowerRelationalNode'>, '<=': <class 'cmp.ast.relational.LowerEqualRelationalNode'>, '==': <class 'cmp.ast.equality.PositiveEqualityNode'>, '>': <class 'cmp.ast.relational.GreaterRelationalNode'>, '>=': <class 'cmp.ast.relational.GreaterEqualRelationalNode'>, '^': <class 'cmp.ast.multiplicative.PowerNode'>, '|': <class 'cmp.ast.logic.OrNode'>}

property **has_errors**

Return type bool

p_additive_expression (*p*)

additive_expression : *multiplicative_expression*
additive_expression '+' multiplicative_expression
additive_expression '-' multiplicative_expression

Return type None

p_and_expression (*p*)

and_expression : *equality_expression*
and_expression '&' equality_expression

Return type None

p_array_element (*p*)

array_element : *expression*
expression_statement

Return type None

p_array_expression (*p*)

array_expression : IDENTIFIER '(' index_expression_list ')'

Return type None

p_array_list (*p*)

array_list : *array_element*
array_list array_element

Return type None

p_assignment_expression (*p*)

assignment_expression : postfix_expression '=' expression

Return type None

p_assignment_statement (*p*)

assignment_statement : assignment_expression eostmt

Return type None

p_clear_statement (*p*)

clear_statement : CLEAR identifier_list eostmt

Return type `None`

`p_comment_statement` (*p*)
comment_statement : COMMENT TCOMMENT

Return type `None`

`p_constant_expression` (*p*)
constant_expression : CONSTANT

Return type `None`

`p_elseif_clause` (*p*)

elseif_clause : *ELSEIF expression statement_list*
elseif_clause ELSEIF expression statement_list

Return type `None`

`p_eostmt` (*p*)

eostmt : ','
';'
NEWLINE

Return type `None`

`p_equality_expression` (*p*)

equality_expression : *relational_expression*
equality_expression EQ_OP relational_expression
equality_expression NE_OP relational_expression

Return type `None`

`p_error` (*p*)

Return type `None`

`p_expression` (*p*)

expression : *or_expression*
expression ':' or_expression

Return type `None`

`p_expression_statement` (*p*)

expression_statement : *eostmt*
expression eostmt

Return type `None`

`p_func_declare` (*p*)

func_declare : *func_declare_lhs*
func_return_list '=' func_declare_lhs
func_declare_invoke_error

Return type `None`

`p_func_declare_invoke_error` (*p*)

func_declare_invoke_error : *func_return_list '='*
func_return_list

Return type `None`

`p_func_declare_lhs` (*p*)

func_declare_lhs : *IDENTIFIER*
IDENTIFIER '(' ')'
IDENTIFIER '(' func_identifier_list ')'

Return type `None`

`p_func_identifier_list` (*p*)

func_identifier_list : *IDENTIFIER*
func_identifier_list ',' IDENTIFIER

Return type `None`

`p_func_return_list` (*p*)

func_return_list : *IDENTIFIER*
'[' func_identifier_list ']'

Return type `None`

`p_func_statement` (*p*)

func_statement : *FUNCTION func_declare eostmt statement_list END*
func_statement_error

Return type `None`

`p_func_statement_error` (*p*)
func_statement_error : FUNCTION error eostmt statement_list END

Return type `None`

`p_global_statement` (*p*)
global_statement : GLOBAL identifier_list eostmt

Return type `None`

`p_identifier_expression` (*p*)
identifier_expression : IDENTIFIER

Return type `None`

`p_identifier_list` (*p*)

identifier_list : *IDENTIFIER*
identifier_list IDENTIFIER

Return type `None`

`p_index_expression` (*p*)

index_expression : *':'*
expression

Return type `None`

`p_index_expression_list` (*p*)

index_expression_list : *index_expression*
index_expression_list ',' index_expression

Return type `None`

`p_iteration_statement` (*p*)

iteration_statement : *WHILE expression statement_list END eostmt*
FOR IDENTIFIER '=' expression statement_list END eostmt

FOR '(' IDENTIFIER '=' expression ')' statement_list END eostmt

Return type `None`

`p_jump_statement` (*p*)

jump_statement : *BREAK eostmt*
RETURN eostmt

Return type `None`

`p_multiplicative_expression` (*p*)

multiplicative_expression : *unary_expression*
multiplicative_expression '*' unary_expression
multiplicative_expression '/' unary_expression
multiplicative_expression '^' unary_expression
multiplicative_expression ARRAY_MUL unary_expression
multiplicative_expression ARRAY_DIV unary_expression
multiplicative_expression ARRAY_RDIV unary_expression
multiplicative_expression ARRAY_POW unary_expression

Return type `None`

`p_or_expression` (*p*)

or_expression : *and_expression*
or_expression '|' and_expression

Return type `None`

`p_postfix_expression` (*p*)

postfix_expression : *primary_expression*
array_expression
postfix_expression TRANSPOSE

Return type `None`

`p_primary_expression` (*p*)

primary_expression : *identifier_expression*
constant_expression
string_literal_expression
'(' expression ')'
'[' ']'
'[' array_list ']'

Return type `None`

`p_relational_expression` (*p*)

relational_expression : *additive_expression*
relational_expression '<' additive_expression
relational_expression '>' additive_expression
relational_expression LE_OP additive_expression
relational_expression GE_OP additive_expression

Return type `None`

`p_selection_statement` (*p*)

selection_statement : *IF expression statement_list END eostmt*
IF expression statement_list ELSE statement_list END eostmt
IF expression statement_list elseif_clause END eostmt
IF expression statement_list elseif_clause ELSE statement_list END eostmt
selection_statement_invoke_error

selection_statement_error

Return type `None`

`p_selection_statement_error` (*p*)
selection_statement_error : IF error

Return type `None`

`p_selection_statement_invoke_error` (*p*)
selection_statement_invoke_error : IF expression statement_list

Return type `None`

`p_statement` (*p*)

statement : *global_statement*
clear_statement
assignment_statement
expression_statement
selection_statement
iteration_statement
jump_statement
func_statement
comment_statement

Return type `None`

`p_statement_list` (*p*)

statement_list : *statement*
statement_list statement
statement_list_error

Return type `None`

`p_statement_list_error` (*p*)
statement_list_error : statement_list error

Return type `None`

`p_string_literal_expression` (*p*)
string_literal_expression : STRING_LITERAL

Return type `None`

`p_translation_unit` (*p*)
translation_unit : statement_list

Return type `None`

`p_unary_expression` (*p*)

unary_expression : *postfix_expression*
unary_operator postfix_expression

Return type `None`

`p_unary_operator` (*p*)

unary_operator : '+'
'-'
'~'

Return type `None`

parse (*text*, *debug_level=False*)

Return type `Any`

precedence = (('right', '-'), ('right', '~'), ('right', '+'))

Module contents

class `cmp.grammar.`**`Lexer`**

Bases: `cmp.helpers.log.LogMixin`

Executive lexer object. Containing primary tokens. This class give next token

D = '[0-9]'

E = '[DdEe][+-]?[0-9]+'

L = '[a-zA-Z_]'

constant = '[0-9]+([DdEe][+-]?[0-9]+)?|[0-9]*"."[0-9]+([DdEe][+-]?[0-9]+)?|[0-9]+"."[0-9]*([D-dEe][+-]?[0-9]+)?'

constant_1 = '[0-9]+([DdEe][+-]?[0-9]+)?'

constant_2 = '[0-9]*"."[0-9]+([DdEe][+-]?[0-9]+)?'

constant_3 = '[0-9]+"."[0-9]*([DdEe][+-]?[0-9]+)?'

identifier = '[a-zA-Z_]([a-zA-Z_]|[0-9])*'

input (*data_*)

Return type `None`

keywords = {'break': 'BREAK', 'clear': 'CLEAR', 'else': 'ELSE', 'elseif': 'ELSEIF', 'end': 'END', 'for': 'FOR', 'function': 'FUNCTION', 'global': 'GLOBAL', 'if': 'IF', 'return': 'RETURN', 'while': 'WHILE'}

literals = ['~', ';', ',', ':', '=', '(', ')', '[', ']', '&', '-', '+', '*', '/', '\\', '>', '<', '|']

states = (('string', 'exclusive'),)

t_ANY_COMMENT (*token_*)

[%]|[n]

Return type `LexToken`

t_ARRAY_DIV = '\\./'

t_ARRAY_MUL = '\\.\\*'

t_ARRAY_POW = '\\.\\^'

t_ARRAY_RDIV = '\\.//'

t_CONSTANT (*token_*)

Return type `LexToken`

t_EQ_OP = '=='

t_GE_OP = '\\>='

t_IDENTIFIER (*token_*)

> **Return type** `LexToken`

t_LE_OP = '<='

t_NEWLINE (*token_*)

> n
>
> **Return type** `LexToken`

t_NE_OP = '(~=)|(!=)'

t_STRING_LITERAL (*token_*)

> '[^'n]*'
>
> **Return type** `LexToken`

t_TRANSPOSE (*token_*)

> **Return type** `LexToken`

t_error (*token_*)

> Error handler lexer
>
> **Return type** `None`

t_ignore_WHITESPACE = '\\s+'

t_string_TCOMMENT = '[^\\n]+'

t_string_error (*token_*)

> Error handler lexer for string state
>
> **Return type** `None`

t_string_ignore = ''

token ( )

> **Return type** `LexToken`

tokens = ('IDENTIFIER', 'CONSTANT', 'STRING_LITERAL', 'LE_OP', 'GE_OP', 'EQ_OP', 'NE_OP', 'ARRAY_MUL', 'ARRAY_POW', 'ARRAY_DIV', 'ARRAY_RDIV', 'TRANSPOSE', 'NEWLINE', 'COMMENT', 'TCOMMENT', 'FOR', 'WHILE', 'BREAK', 'IF', 'ELSE', 'ELSEIF', 'END', 'FUNCTION', 'RETURN', 'GLOBAL', 'CLEAR')

transpose = "'|\\.'"

transpose_1 = "'"

transpose_2 = "\\.'"

class `cmp.grammar.`**Parser** (*lexer=<class 'cmp.grammar.lexer.Lexer'>, yacc_debug=False*)

> Bases: `object`
>
> Executive parser object. Containing primary reduce rules. This class build AST
>
> **errors** ( )
>
> > **Return type** `Iterator[str]`

handlers = {'!=': <class 'cmp.ast.equality.NegativeEqualityNode'>, '&': <class 'cmp.ast.logic.AndNode'>, '*': <class 'cmp.ast.multiplicative.MultiplyNode'>, '+': <class 'cmp.ast.additive.PlusNode'>, '-': <class 'cmp.ast.additive.MinusNode'>, '.*': <class 'cmp.ast.multiplicative.ArrayMulNode'>, './': <class 'cmp.ast.multiplicative.ArrayDivNode'>, './/': <class 'cmp.ast.multiplicative.ArrayRDivNode'>, '.^': <class 'cmp.ast.multiplicative.ArrayPowerNode'>, '/': <class 'cmp.ast.multiplicative.DivideNode'>, '<': <class 'cmp.ast.relational.LowerRelationalNode'>, '<=': <class 'cmp.ast.relational.LowerEqualRelationalNode'>, '==': <class 'cmp.ast.equality.PositiveEqualityNode'>, '>': <class 'cmp.ast.relational.GreaterRelationalNode'>, '>=': <class 'cmp.ast.relational.GreaterEqualRelationalNode'>, '^': <class 'cmp.ast.multiplicative.PowerNode'>, '|': <class 'cmp.ast.logic.OrNode'>}

property **has_errors**

Return type `bool`

p_additive_expression (*p*)

additive_expression : *multiplicative_expression*
additive_expression '+' multiplicative_expression
additive_expression '-' multiplicative_expression

Return type `None`

p_and_expression (*p*)

and_expression : *equality_expression*
and_expression '&' equality_expression

Return type `None`

p_array_element (*p*)

array_element : *expression*
expression_statement

Return type `None`

p_array_expression (*p*)

array_expression : IDENTIFIER '(' index_expression_list ')'

Return type `None`

p_array_list (*p*)

array_list : *array_element*
array_list array_element

Return type `None`

p_assignment_expression (*p*)

assignment_expression : postfix_expression '=' expression

Return type `None`

p_assignment_statement (*p*)

assignment_statement : assignment_expression eostmt

Return type `None`

p_clear_statement (*p*)

clear_statement : CLEAR identifier_list eostmt

Return type `None`

p_comment_statement (*p*)

comment_statement : COMMENT TCOMMENT

Return type `None`

`p_constant_expression` (*p*)
constant_expression : CONSTANT

Return type `None`

`p_elseif_clause` (*p*)

elseif_clause : *ELSEIF expression statement_list*
elseif_clause ELSEIF expression statement_list

Return type `None`

`p_eostmt` (*p*)

eostmt : *','*
';'
NEWLINE

Return type `None`

`p_equality_expression` (*p*)

equality_expression : *relational_expression*
equality_expression EQ_OP relational_expression
equality_expression NE_OP relational_expression

Return type `None`

`p_error` (*p*)

Return type `None`

`p_expression` (*p*)

expression : *or_expression*
expression ':' or_expression

Return type `None`

`p_expression_statement` (*p*)

expression_statement : *eostmt*
expression eostmt

Return type `None`

`p_func_declare` (*p*)

func_declare : *func_declare_lhs*
func_return_list '=' func_declare_lhs
func_declare_invoke_error

Return type `None`

`p_func_declare_invoke_error` (*p*)

func_declare_invoke_error : *func_return_list '='*
func_return_list

Return type `None`

`p_func_declare_lhs` (*p*)

func_declare_lhs : *IDENTIFIER*
IDENTIFIER '(' ')'
IDENTIFIER '(' func_identifier_list ')'

Return type `None`

`p_func_identifier_list` (*p*)

func_identifier_list : *IDENTIFIER*
func_identifier_list ',' IDENTIFIER

Return type `None`

`p_func_return_list` (*p*)

func_return_list : *IDENTIFIER*
'[' func_identifier_list ']'

Return type `None`

`p_func_statement` (*p*)

func_statement : *FUNCTION func_declare eostmt statement_list END*
func_statement_error

Return type `None`

`p_func_statement_error` (*p*)
func_statement_error : FUNCTION error eostmt statement_list END

Return type `None`

`p_global_statement` (*p*)
global_statement : GLOBAL identifier_list eostmt

Return type `None`

`p_identifier_expression` (*p*)
identifier_expression : IDENTIFIER

Return type `None`

`p_identifier_list` (*p*)

identifier_list : *IDENTIFIER*
identifier_list IDENTIFIER

Return type `None`

`p_index_expression` (*p*)

index_expression : *':'*
expression

Return type `None`

`p_index_expression_list` (*p*)

index_expression_list : *index_expression*
index_expression_list ',' index_expression

Return type `None`

`p_iteration_statement` (*p*)

iteration_statement : *WHILE expression statement_list END eostmt*
FOR IDENTIFIER '=' expression statement_list END eostmt
FOR '(' IDENTIFIER '=' expression ')' statement_list END eostmt

Return type `None`

`p_jump_statement` (*p*)

jump_statement : *BREAK eostmt*
RETURN eostmt

Return type `None`

`p_multiplicative_expression` (*p*)

multiplicative_expression : *unary_expression*
multiplicative_expression '*' unary_expression
multiplicative_expression '/' unary_expression
multiplicative_expression '^' unary_expression
multiplicative_expression ARRAY_MUL unary_expression
multiplicative_expression ARRAY_DIV unary_expression
multiplicative_expression ARRAY_RDIV unary_expression
multiplicative_expression ARRAY_POW unary_expression

Return type `None`

`p_or_expression` (*p*)

or_expression : *and_expression*
or_expression '|' and_expression

Return type `None`

`p_postfix_expression` (*p*)

postfix_expression : *primary_expression*
array_expression
postfix_expression TRANSPOSE

Return type `None`

`p_primary_expression` (*p*)

primary_expression : *identifier_expression*
constant_expression
string_literal_expression
'(' expression ')'
'[' ']'
'[' array_list ']'

Return type `None`

`p_relational_expression` (*p*)

relational_expression : *additive_expression*
relational_expression '<' additive_expression
relational_expression '>' additive_expression
relational_expression LE_OP additive_expression
relational_expression GE_OP additive_expression

Return type `None`

`p_selection_statement` (*p*)

selection_statement : *IF expression statement_list END eostmt*
IF expression statement_list ELSE statement_list END eostmt
IF expression statement_list elseif_clause END eostmt
IF expression statement_list elseif_clause ELSE statement_list END eostmt
selection_statement_invoke_error
selection_statement_error

Return type `None`

p_selection_statement_error (*p*)
selection_statement_error : IF error

Return type `None`

p_selection_statement_invoke_error (*p*)
selection_statement_invoke_error : IF expression statement_list

Return type `None`

p_statement (*p*)

statement : *global_statement*
clear_statement
assignment_statement
expression_statement
selection_statement
iteration_statement
jump_statement
func_statement
comment_statement

Return type `None`

p_statement_list (*p*)

statement_list : *statement*
statement_list statement
statement_list_error

Return type `None`

p_statement_list_error (*p*)
statement_list_error : statement_list error

Return type `None`

p_string_literal_expression (*p*)
string_literal_expression : STRING_LITERAL

Return type `None`

p_translation_unit (*p*)
translation_unit : statement_list

Return type `None`

p_unary_expression (*p*)

unary_expression : *postfix_expression*
unary_operator postfix_expression

Return type `None`

p_unary_operator (*p*)

unary_operator : '+'
'-'
'~'

Return type `None`

parse (*text, debug_level=False*)

Return type `Any`

precedence = (('right', '-'), ('right', '~'), ('right', '+'))

Helpers package

Subpackages

Server package

Submodules

Server setup

Tcp client module

class `cmp.helpers.server.tcp_client.`**`TCPClient`** (**args, **kwargs*)

Bases: `argparse.ArgumentParser`

Simple TCP client for sending in MATLAB compiler

`execute` ( )

Start TCP client in CLI

Return type `None`

`cmp.helpers.server.tcp_client.`**`main`** ( )

Return type `None`

Tcp server module

class `cmp.helpers.server.tcp_server.`**`TCPServer`** (*consumer, host=None, port=None*)

Bases: `cmp.helpers.log.ServerLog`

Server for service matlab compiler hostname - IP address or domen name local machine where is server will run port - the port that the server will listen to consumer - function what will give result str -> str

async **`execute`** ( )

Start TCP server

Return type `None`

Module contents

class `cmp.helpers.server.`**`TCPClient`** (**args, **kwargs*)

Bases: `argparse.ArgumentParser`

Simple TCP client for sending in MATLAB compiler

`execute` ( )

Start TCP client in CLI

Return type `None`

class `cmp.helpers.server.`**`TCPServer`** (*consumer, host=None, port=None*)

Bases: `cmp.helpers.log.ServerLog`

Server for service matlab compiler hostname - IP address or domen name local machine where is server will run port - the port that the server will listen to consumer - function what will give result str -> str

async **`execute`** ( )

Start TCP server

Return type `None`

Submodules

Camel to snake module

`cmp.helpers.camel_to_snake.`**`camel_to_snake`** (*name*)

Convert camel case to snake case

Return type `str`

Colors module

class `cmp.helpers.colors.`**`colors`**

Bases: `object`

`BOLD` = '\x1b[1m'

`ENDC` = '\x1b[0m'

`FAIL` = '\x1b[91m'

`HEADER` = '\x1b[95m'

`OKBLUE` = '\x1b[94m'

`OKCYAN` = '\x1b[96m'

`OKGREEN` = '\x1b[92m'

`UNDERLINE` = '\x1b[4m'

`WARNING` = '\x1b[93m'

Exceptions module

exception `cmp.helpers.exceptions.`**`BadInputError`** (*text=''*)

Bases: `Exception`

Exceptions of None root

Logger module

class `cmp.helpers.log.`**`LogMixin`**

Bases: `object`

Class mixin for produce logger entire class

property **`logger`**

Return type `Logger`

class `cmp.helpers.log.`**`ServerLog`** (*name='server'*)

Bases: `object`

Custom logger for TCP server

Pattern singleton module

class `cmp.helpers.singleton.`**`Singleton`** (**args, **kwargs*)

Bases: `object`

Base class for release pattern Singleton

Module contents

exception `cmp.helpers.`**`BadInputError`** (*text=''*)

Bases: `Exception`

Exceptions of None root

class `cmp.helpers.`**`LogMixin`**

Bases: `object`

Class mixin for produce logger entire class

property **`logger`**

Return type `Logger`

class `cmp.helpers.`**`Singleton`** (**args*, ***kwargs*)

Bases: `object`

Base class for release pattern Singleton

`cmp.helpers.`**`camel_to_snake`** (*name*)

Convert camel case to snake case

Return type `str`

class `cmp.helpers.`**`colors`**

Bases: `object`

`BOLD` = **'\x1b[1m'**

`ENDC` = **'\x1b[0m'**

`FAIL` = **'\x1b[91m'**

`HEADER` = **'\x1b[95m'**

`OKBLUE` = **'\x1b[94m'**

`OKCYAN` = **'\x1b[96m'**

`OKGREEN` = **'\x1b[92m'**

`UNDERLINE` = **'\x1b[4m'**

`WARNING` = **'\x1b[93m'**

Traverse package

Submodules

Traverse AST module

class `cmp.traverse.traverse_ast.`**`Visitor`** (*numpy_mode=False*, *filename=None*)

Bases: `object`

Walk through the generated AST and translating it to Python code in the specified file Recursive walking in tree invoke methods _visit + NameNode numpy_mode - if activated, ordinary operation will be translated

in numpy operations

filename - name of file where will be written python code _depth - inner variable for count ident _stack - store for return value of function

`keywords` = **{'diag': 'np.diag', 'eye': 'np.eye', 'ones': 'np.ones', 'rand': 'np.random', 'zeros': 'np.zeros'}**

property **`py_tab`**

Return type `str`

`tabulate_expr` (*expr*)

Shift expression on python tab and filtering entered expression on empty symbol or symbol

Return type `str`

`traverse_ast` (*root*)

Main function for traverse and printing translated code. :param: root: FileAST - root of AST :rtype: `Optional[str]` :return: python code or None if a file was specified when creating an object

Module contents

1.1.2 Module contents

- genindex
- modindex
- search

Python Module Index