B 演習 (言語処理系演習) 資料
mini-Python 文法

田浦

2006/10/30

```
#
# mini-Python 文法
#
```

token (字句) の定義:

```
    identifier | stringliteral | integer | floatnumber
  | INDENT | DEDENT | NEWLINE
  | None | or | and | not | is | in
  | pass | return | continue | break
  | if | elif | else | while | for | def
  | [ | ] | { | } | ( | )
  | . | , | :
  | - | + | * | / | % | & | | | ~ | ^
  | !=
  | < | << | <= | <> | > | >> | >= | = | ==
```

3 行目以降に現れる字句 (たとえば or や /) はまさに書かれた文字の列がそのまま各字句の定義になる．2 行目の NEWLINE は改行，INDENT/DEDENT は字下げレベルが変ったときに人工的に生成される字句で，詳しくは資料を参照せよ．

identifier (変数などの名前), stringliteral (文字列リテラル), integer (整数リテラル), floatnumber (浮動小数点リテラル) は，以下で正規表現で定義されている．

正規表現:

a|b	a または b
a*	a を 0 回以上繰り返し
a+	a を 1 回以上繰り返し
a?	a が 0 回または 1 回

```
#
# 変数などの名前
#
identifier ::=
             (letter | '_') (letter | digit | '_')*

letter ::=
             lowercase | uppercase

lowercase ::=
             'a'...'z'

uppercase ::=
             'A'...'Z'

digit ::=
             '0'...'9'

#
# 文字列リテラル
#
stringliteral ::=
              '"' shortstringitem* '"'

shortstringitem ::=
             shortstringchar | escapeseq

shortstringchar ::=
```

```
             <any ASCII character except '\' or newline or the quote>
escapeseq ::=
             '\' <any ASCII character>

#
# 整数リテラル
#
integer ::=
             nonzerodigit digit* | '0'
nonzerodigit ::=
             '1'...'9'

#
# 浮動小数点リテラル. 以下は皆 valid な例  1.2    123.     .538
# . 一文字は×なので注意
#
floatnumber ::=
             digit+ '.' digit* | '.' digit+


#
# "(" expression_list_with_comma ")" について
# expression_list_with_comma は,
#  式を 0 個以上カンマで区切りながら並べ,
#  最後にオプションとして余分なカンマがあっても良い,
#  というものである.
#  ただし式の数が 0 のときは, 最後の余分なカンマはあってはいけない
#  (前方の定義参照).
#
# 以下はどれも正しい atom の例である
# ()
# (1)
# (1,)
# (1,2)
# (1,2,)
# --- そしてこのうち, (1) 以外はどれもタプルである.
# --- (1) は 1 という式を括弧にくくったものである.
# 1 要素のタプルを作るのに, (1,) という表現を使うことに注意
#
atom ::=
             identifier | literal | list_display | dict_display
           | '(' expression_list_with_comma ')'
expression_list_with_comma ::=
             [ expression ( ',' expression )* [','] ]

#
# リテラル (文字列, 整数, 浮動小数点数)
#
literal ::=
             stringliteral | integer | floatnumber


#
# リスト式  [ 式, 式, ... ]
# タプルと文法を統一するために, 最後の余分な , を許す.
# この , はあってもなくても,
# 意味は変わらない
#
list_display ::=
             '[' expression_list_with_comma ']'

#
# 辞書式  { 式 : 式, 式 : 式, ... }
#
dict_display ::=
             '{' key_datum_list '}'
key_datum_list ::=
             [ key_datum (',' key_datum)* ]
key_datum ::=
             expression ':' expression
```

```
#
# - atom.f は attref 式．mini-Python ではその直後に atom.f(x, y, z)
#   と続く場合のみが許されるが,
#   このことは以下の文法には現れていない (parser が別途チェックしている).
#   そして，atom.f(x, y, z) は f(atom, x, y, z) と等価としている．
# - atom[expr] は 構文木では subscript 式に対応している．
#   atom[x,y,z] と来た場合，expr がタプル (x,y,z) である
#   とみなす．つまり， atom[x,y,z] = atom[(x,y,z)]
# - atom(x, y, z) は関数呼び出し．構文木では call
#
primary ::=
             atom ’.’ identifier
               | ’[’ expression_list_with_comma ’]’
               | ’(’ expression_list_with_comma ’)’ )*

#
# いわゆる単項演算子．
# 以下に沢山種類があるのは，演算子の優先順位を反映するため．
#
u_expr ::=
             primary | ’-’ u_expr | ’+’ u_expr | ’~’ u_expr

#
# 2 項演算子多数
#
m_expr ::=
             u_expr ( ( ’*’ | ’/’ | ’%’ ) u_expr )*
a_expr ::=
             m_expr ( ( ’+’ | ’-’ ) m_expr )*
shift_expr ::=
             a_expr ( ( ’<<’ | ’>>’ ) a_expr )*
and_expr ::=
             shift_expr ( ’&’ shift_expr )*
xor_expr ::=
             and_expr ( ’^’ and_expr )*
or_expr ::=
             xor_expr ( ’|’ xor_expr )*
#
#
comparison ::=
             or_expr [ comp_operator or_expr ]
comp_operator ::=
             ’<’ | ’>’ | ’==’ | ’>=’ | ’<=’ | ’!=’
                | ’is’ [’not’] | [’not’] ’in’
not_test ::=
             comparison | ’not’ not_test
and_test ::=
             not_test ( ’and’ not_test )*
or_test ::=
             and_test ( ’or’ and_test )*
expression ::=
             or_test
#
# 式文
#
expression_stmt ::=
             expression
#
# 代入文
#
assignment_stmt ::=
             target ’=’ expression
```

```
#  代入文の左辺は「変数」または「subscription」 (a[x] etc.)
target ::=
             identifier
                | subscription

#
#
#
subscription ::=
        primary '[' expression_list_with_comma ']'

pass_stmt ::=
             'pass'

#
#
#
return_stmt ::=
             'return' [expression_list_with_comma]

del_stmt ::=
             'del' subscription

break_stmt ::=
             'break'

continue_stmt ::=
             'continue'

global_stmt ::=
             'global' identifier

#
#
#
print_stmt ::=
             'print' expression_list_with_comma

suite ::=
             NEWLINE INDENT statement+ DEDENT

statement ::=
             expression_stmt NEWLINE
              | assignment_stmt NEWLINE
              | pass_stmt NEWLINE
              | return_stmt NEWLINE
              | del_stmt NEWLINE
              | break_stmt NEWLINE
              | continue_stmt NEWLINE
              | global_stmt NEWLINE
              | print_stmt NEWLINE
              | if_stmt
              | while_stmt
              | for_stmt
              | funcdef

if_stmt ::=
             'if' expression ':' suite
                ( 'elif' expression ':' suite )*
                ['else' ':' suite]

while_stmt ::=
             'while' expression ':' suite

for_stmt ::=
             'for' identifier 'in' expression ':' suite

funcdef ::=
             'def' funcname '(' parameter_list ')' ':' suite

parameter_list ::=
             [ identifier (',' identifier)* ]

funcname ::=
             identifier

file_input ::=
             (NEWLINE | statement)*
```