Iris 言語仕様

はらけん

2007 年 10 月 11 日

Contents

1 Iris 紹介

1-1 Iris とは

Iris は C++ 言語によって記述されたインタプリタ方式のスクリプト言語である．Iris は，Python を基軸に Perl や Java の影響を受けた取り合わせ言語である．スクリプト言語としての強力さを確保しつつ，文法的に美しく記述しやすい言語を目指している．機能としては，リスト，ハッシュ，関数，クラス，オブジェクト，例外処理，スレッド，正規表現などを備えている．モジュールの組み込みをサポートするので，プログラムの再利用性や拡張性にも優れる．

なお，Iris とは「い（*i*）ささかご都合主義の，論（*r*）理的でわかりやすく，い（*i*）ささかご都合主義の，ス（*s*）クリプト言語」の略である．

1-2 Iris 哲学

%%% 言語の哲学を書く．

1-3 Iris の機能

%%% 後日詳しく書くので，とりあえずは実装予定のメモ．

文字列，リスト，ハッシュ，関数，クラス，オブジェクト，例外処理，スレッド，正規表現，イベントリスナ，ガベージコレクタを実装する予定．

とりあえず，（ほとんど書き換えることになるが）mini-Python を発展させる形で，基本的なもの（文字列，リスト，ハッシュ，関数，基本的な演算）を実装する．そのあとでクラスを実装する．おそらくクラスが実装できればあとの機能を実装するはかなり立てやすくなる．例外処理とスレッドを実装する．基本的なモジュールを整える．イベントリスナは実装するかどうかは未定．最後に時間があればガベージコレクタを整える．正規表現は，C 言語のライブラリをラップする形にする．

1-4 Iris で Hello,world!

`Hello,world!`は以下のように出力する：

```
print("Hello,world!")
```

1-5 Iris の実行方法

プログラムを `tulip.iris` という名前で用意したら，`iris` コマンドで実行する：

```
$ iris tulip.iris
```

1-6 コマンドラインオプション

%%% 具体的に何を用意するかは未定.

2 文法定義

Iris は式 (expr) と文 (stmt) からなる言語である.
BNF 記法によって構文規則を以下のように定義する：

```
###########要素###########

###要素
atom ::= identifier | literal | list | hash | null

#識別子
identifier ::= ( letter | '_' ) ( letter | digit | '_' )*
letter ::= 'a' | 'b' | ... | 'z' | 'A' | 'B' | ... | 'Z'
digit ::= '0' | '1' | ... | '9'

#リテラル
literal ::= string | integer | float

#文字列
string ::= '"' ( character | escapesequence )* '"'
character ::= <any ASCII character ^ ( '\' | '"' )>
escapeseqence ::= '\' <any ASCII character>

#数
integer ::= dec_integer | oct_integer | hex_integer | bin_integer

#10 進数
dec_integer ::= nonzero_digit digit* | '0'
nonzero_digit ::= '1' | '2' | ... | '9'

#8 進数
oct_integer ::= '0' oct_digit+
oct_digit ::= '0' | '1' | ... | '7'

#16 進数
hex_integer ::= '0' ( 'x' | 'X' ) hex_digit+
```

```
hex_digit ::= '0' | '1' | ... | '9' | 'a' | 'b' | ... | 'f'
            | 'A' | 'B' | ... | 'F'

#2 進数
bin_integer ::= '0' ( 'b' | 'B' ) bin_digit+
bin_digit ::= '0' | '1'

#浮動小数点数
float ::= dec_integer '.' digit+

#リスト
list ::= '[' csv_expr ']'

#ハッシュ
hash ::= '{' [ expr '->' expr ( ',' expr '->' expr )* ] '}'

#null
null ::= 'null'

###########演算子###########

#一次語
primary ::= atom ( '.' identifier | '[' csv_expr ']'
                                  | '(' csv_expr ')' )*

#ユニット
unit ::= primary | '(' unit ')'

#単項演算子
unary_expr ::= unit | '-' unary_expr
             | '~' unary_expr | '!' unary_expr
             | 'new' identifier '(' csv_expr ')'

#指数演算
power_expr ::= unary_expr ( '**' unary_expr )*

#乗算，除算，剰余
mul_expr ::= power_expr ( ( '*' | '/' | '%' ) power_expr )*

#加算，減算
add_expr ::= mul_expr ( ( '+' | '-' ) mul_expr )*

#シフト演算
shift_expr ::= add_expr ( ( '<<' | '>>' ) add_expr )*
```

#**ビット論理積**
```
bitand_expr ::= shift_expr ( '&' shift_expr )*
```

#**ビット排他的論理和**
```
bitxor_expr ::= bitand_expr ( '^' bitand_expr )*
```

#**ビット論理和**
```
bitor_expr ::= bitxor_expr ( '|' bitxor_expr )*
```

#**比較演算**
```
compare_expr ::= bitor_expr [ comp_operator bitor_expr ]
comp_operator ::= '<' | '>' | '<=' | '>='
                | '==' | '!=' | 'in' | 'not' 'in'
```

#**論理積**
```
and_expr ::= compare_expr ( '&&' compare_expr )*
```

#**論理和**
```
or_expr ::= and_expr ( '||' and_expr )*
```

#**範囲演算子**
```
range_expr ::= or_expr [ '..' or_expr ]
```

#**三項演算子**
```
triple_expr ::= range_expr [ '?' range_expr ':' range_expr ]
```

###**式**
```
expr ::= triple_expr
```

##########**いろいろ**##########

#**式リスト**
```
csv_expr ::= [ expr ( ',' expr )* ]
```

#**識別子リスト**
```
csv_identifier ::= [ identifier ( ',' identifier )* ]
```

#**単独左辺値**
```
one_target ::= identifier | subscript | attribute
```

#subscript **式**
```
subscript ::= primary '[' csv_expr ']'
```

#**オブジェクト参照**
```
attribute ::= primary '.' identifier
```

#**関数呼び出し**
```
call ::= primary '(' csv_expr ')'

##########文##########

###文
stmt ::= expr_stmt NEWLINE
       | assignment_stmt NEWLINE
       | augment_stmt NEWLINE
       | increment_stmt NEWLINE
       | pass_stmt NEWLINE
       | null_stmt NEWLINE
       | return_stmt NEWLINE
       | break_stmt NEWLINE
       | continue_stmt NEWLINE
       | if_stmt
       | while_stmt
       | select_stmt
       | for_stmt
       | def_stmt
       | try_stmt
       | throw_stmt NEWLINE
       | class_stmt
       | listener_stmt
       | use_stmt NEWLINE

#式文
expr_stmt ::= expr

#代入文
assignment_stmt ::= ( target '=' )+ expr
target ::= one_target | '[' one_target ( ',' one_target )* ']'

#累算代入文
augment_stmt ::= one_target ( '+=' | '-=' | '*=' | '/=' | '%='
               | '**=' | '<<=' | '>>=' | '&=' | '^=' | '|=' ) expr

#increment,decrement 文
increment_stmt ::= one_target ( '++' | '--' )

#pass 文
pass_stmt ::= 'pass'

#null 文
```

```
null_stmt ::= ''

#return 文
return_stmt ::= 'return' csv_expr

#break 文
break_stmt ::= 'break' ( ',' 'break' )*

#continue 文
continue_stmt ::= 'continue'

#ブロック
suite ::= NEWLINE+ INDENT stmt+ DEDENT

#if 文
if_stmt ::= 'if' expr suite ( 'elif' expr suite )*
            [ 'else' suite ]

#while 文
while_stmt ::= 'while' expr suite

#select 文
select_stmt ::= 'select' expr NEWLINE+ INDENT
              ( 'case' expr suite )* [ 'else' suite ] DEDENT

#for 文
for_stmt ::= 'for' identifier 'in' list suite

#def 文
def_stmt ::= 'def' identifier '(' csv_identifier ')' suite

#try 文
try_stmt ::= 'try' suite ( 'catch' identifier suite )*
             [ 'else' suite ]

#throw 文
throw ::= 'throw'

#class 定義
class_stmt ::= 'class' identifier suite

#listener 定義
listener_stmt ::= 'listener' identifier case_suite
case_suite ::= NEWLINE+ INDENT ( 'case' identifier suite )*
               [ 'default' suite ] DEDENT
```

#use **文**
```
use_stmt ::= 'use' identifier
```

##########**ファイル**##########

###**ファイル**
```
file ::= stmt*
```

3 データ

3-1 データの種類

データには，整数，浮動小数点数，文字列，リスト，ハッシュ，null，関数，クラス，オブジェクト，リスナがある．整数，浮動小数点数，文字列，null はプリミティブ型であり，リスト，ハッシュ，関数，クラス，オブジェクト，リスナは参照型である．

変数に代入可能なのはデータのみである．

3-2 整数

その処理系に実装されている C 言語の `int` 型で表現可能な数である．10 進表記，2 進表記，8 進表記，16 進表記が可能である．たとえば，次の例はどれも正しい：

```
5   -11111   0777   0x1234   0b01010101
```

`+17` のような無駄な正符号は許されない．

3-3 浮動小数点数

C 言語の `double` 型で表現可能な数である．たとえば，次の例はどれも正しい：

```
3.14159   0.25
```

ただし，`0.12` を `.12` と書いたり，`3.0` を `3.` と書いたり，`3.0` を `+03.0` と書くような表記は許されない．また，`e` や `E` を用いた指数表記には対応していないが，これは`**`演算子を使えばよい：

```
print(4*10**(-5)) # 0.00004
```

3-4 文字列

¨と¨で囲まれた領域に任意の ASCII 文字を含めることができる．たとえば，次の例はどれも正しい：

```
"tulip"
"cosmos
pansy
rose"
```

2 つ目の例は 3 行に渡る文字列である．文字列の中では，"と \ と { と } だけは特別扱いされるため，これらの文字を出力するには，その文字の直前に \ を置いてエスケープする必要がある．

エスケープシーケンスとしては以下を使用可能である：

\n（改行），\t（タブ），\r（リターン），\f（改ページ），\b（バックスペース），\v（垂直タブ），\a（ベル）

また，文字列中で { と } で囲まれた文字列は変数展開される：

```
list = [1,2,3]
string = "tulip"
print("list{list}string{string}") # list[ 1, 2, 3]stringtulip
```

変数展開は「完全に」行われる．すなわち，たとえばリストの要素としてリストが入っている場合，要素として含まれている方のリストも展開対象になる．{ と } で囲まれた文字列に相当する変数が存在しない場合にはエラーになる．

i 番目の文字列にアクセスするためには，**文字列名**[i] を用いる．文字型は存在しないので，返されるのは 1 文字からなる文字列である．コンマ区切りで複数のインデックスを指定することにより，該当位置にある文字を連結した文字列を作成できる：

```
string = "tulip"
print(string[0],string[0,1],string[4,1,0]) # t tu put
```

3-5 リスト

リストは，要素をコンマで区切って並べたものを [と] で囲んだものである．リストには任意の要素を入れることが可能である．たとえば，次の例はどれも正しい：

```
[1,2,3]   ["tulip",[[1,2],3],4]   []
```

リストの最後の要素の後に余分なコンマを付けることは許されない．

また，リストの中にリストを含める場合などでは，含められるリストはその場で展開されることなくリストのまま格納される：

```
list1 = [1,2]
```

```
list2 = [list1,3] # [1,2,3] ではなく，[[1,2],3] となる
```

なおリスト型は，内部的には C++ の STL の<deque>型として実装されている．双方向リストとして実装されているわけではない．すなわち，先頭への要素の追加（shift），先頭からの要素の削除（unshift），末尾への要素の追加（push），末尾からの要素の削除（pop），i 番目の要素の参照を $O(1)$ で実現する．一方で，i 番目への要素の挿入や i 番目の要素の削除などは $O(n)$ になる．

リストのインデックスは 0 から始まる．リストの i 番目の要素へのアクセスには，**リスト名**[i] を用いる．i は-**（リストのサイズ）**<=i<**（リストのサイズ）**を満たしていないとエラーになる．i に負数を指定した場合には，**リスト名**[i+**（リストのサイズ）**] が参照される：

```
list = [0,1,2]
print(list[0],list[1],list[2],list[-1],list[-2],list[-3])
   # 0 1 2 2 1 0
```

多次元配列も容易に表現できる：

```
list = [[1,2],"tulip",[[[]]]]
print(list[0][1],list[1][4],list[2][0][0]) # 2 p []
```

また，要素へのアクセスにはリストスライスを利用することができる．[と] で囲まれた領域にコンマ区切りで記述されたインデックスの要素が参照され，それらの集合が得られる．得られるのは要素の集合であり，リストではない：

```
list = [0,1,2,3,4,5,6,7]
print(list[0,1,2],[list[-2,6]]) # 0 1 2 [ 6, 6]
```

後で述べる範囲演算子を用いるとより柔軟なリストスライスが可能になる．

3-6 ハッシュ

ハッシュは，キーと値を->で対応させたものをコンマで区切って並べ，{ と } で囲んだものである．キーと値には任意の型を入れられるが，ハッシュ内でキーは一意である必要がある．値には重複があっても良い．たとえば，次の例はどれも正しい：

```
{1->2,"tulip"->[1,2],[]->[1,2]}   {}
```

キーに対応する値を得るには**ハッシュ名**[**キー**] とする．キーに対応する値が存在しない場合にはエラーになる．

また，ハッシュスライスを利用することもできる：

```
hash = {"tulip"->1,"cosmos"->2,"pansy"->[1,2]}
print(hash["tulip","pansy"]) # 1 [ 1, 2]
```

ハッシュのキーとして参照型を使用した場合には注意が必要である．具体例を示す：

```
list = [1,2]
hash = {list->3}
another_list = [1,2]
print(hash[another_list]) # 3
list[1] = 10
print(hash[list]) # エラー!
```

hash={[1,2]->3} の意味は，「1 と 2 が並んだリストに対応する値が 3」なので，このような結果になる．

3-7 null

null には null という値のみが存在する．null は「値がない」ことを意味する．たとえば，return 文に戻り値を指定しなかった場合 null が返される．ただし，null は正確には「本当に値が存在しないこと」ではなく，「値が存在しないという値」を意味する：

```
print(["tulip","cosmos"][3]) # null ではなく，エラー！
```

3-8 関数

関数定義に関してはあとで述べる．定義された関数を他の変数に代入して使用することができる．関数は，**関数名（引数リスト）**の形式で呼び出す：

```
def tulip(n)
   return n+1

cosmos = tulip
print(tulip(3),cosmos(3)) # 4 4
```

3-9 クラス

クラス定義に関してはあとで述べる．定義されたクラスを他の変数に代入して使用することができる：

```
class Tulip
   x = 0 # クラス変数を初期化

Cosmos = Tulip
Tulip.x = 5
print(Tulip.x,Cosmos.x) # 5 5
```

3-10 オブジェクト

オブジェクトの詳細はあとで述べる．オブジェクトは new 演算子を使って生成する．生成されたオブジェクトを他の変数に代入して使用することができる：

```
class Tulip
   def new()
      x = 0 # インスタンス変数を初期化

tulip = new Tulip() # tulip は Tulip クラスのオブジェクト
cosmos = tulip
tulip.x = 5
print(tulip.x,cosmos.x) # 5 5
```

3-11 リスナ

%%% リスナを実装するかどうかは未定.

リスナ定義はあとで述べる．定義されたリスナを他の変数に代入して使用することができる：

```
listener Tulip
   ... # リスナの内容を記述

Cosmos = Tulip
```

4 演算子

4-1 規則

4-1-1 優先順位

演算子の優先順位は以下の通りである．ここで，同一行に書かれた演算子たちは同順位である：

```
( ) # 演算順位を変えるための括弧
. [ ] ( ) # 関数呼び出しの括弧
_ ~ !
**
* / %
+ -
&
^
|
< > <= >= == !=
&&
```

```
||
..
new
```

4-1-2 結合規則

2 項演算子の結合規則は全て左である.

4-1-3 ブール値

真偽値評価では，0，null，空文字列が偽であり，それ以外は真として扱われる．ブール型に相当するデータは存在せず，比較演算および否定演算の結果は，真は整数の 1 で，偽は整数の 0 で返される.

4-1-4 型変換

整数と浮動小数点数の間では自動で型変換が行われる．2 項演算において，一方のオペランドが浮動小数点数ならば他方のオペランドも浮動小数点数に変換されて演算が行われる．また，整数がオーバーフローした場合，浮動小数点数に型変換される.

4-2 –（負符号単項演算子）

オペランドは整数もしくは浮動小数点数.
符号を負にする.

4-3 ~（ビット反転演算子）

オペランドは整数.
ビットを反転させる.

4-4 !（否定演算子）

オペランドは任意である.
オペランドが真である場合には偽を返し，偽である場合には真を返す.

4-5 **（指数演算子）

（左オペランド，右オペランド）の組み合わせは，（整数，整数），（整数，浮動小数点数），（浮動小数点数，整数），（浮動小数点数，浮動小数点数）.

指数演算を行う：

```
print(5**3,2.1**2.1) # 125 4.7496
```

4-6 *（乗算演算子）

（左オペランド，右オペランド）の組み合わせは，(整数，整数)，(整数，浮動小数点数)，(浮動小数点数，整数)，(浮動小数点数，浮動小数点数)，(文字列，整数 [非負整数])，(リスト，整数 [非負整数])．

整数と浮動小数点の組み合わせでは乗算を行う．文字列，リストを左オペランドにすると，左オペランドを右オペランドに指定された非負整数回だけ繰り返したものが生成される：

```
print(3*5.4) # 16.2
print("tulip"*3) # tuliptuliptulip
print(["cosmos",[1,2]]*0) # []
```

4-7 /（除算演算子）

（左オペランド，右オペランド）の組み合わせは，(整数，整数 [非 0])，(整数，浮動小数点数 [非 0])，(浮動小数点数，整数 [非 0])，(浮動小数点数，浮動小数点数 [非 0])．

除算を行う．(整数，整数 [非 0]) の場合には，割った結果が切り捨てられて整数となる．

4-8 %（剰余演算子）

（左オペランド，右オペランド）の組み合わせは，(整数，整数 [非 0])，(整数，浮動小数点 [非 0])，(浮動小数点数，整数 [非 0])，(浮動小数点数，浮動小数点数 [非 0])．

左オペランドを右オペランドで割った余りを返す．どちらか一方の負数にオペランドを指定したときの動作には注意する：

```
print(7%3,6.29%3.14,-3.0%0.7) # 1 0.01 -0.2
```

4-9 +（加算演算子）

（左オペランド，右オペランド）の組み合わせは，(整数，整数)，(整数，浮動小数点数)，(浮動小数点数，整数)，(浮動小数点数，浮動小数点数)，(文字列，文字列)，(リスト，リスト)．

整数と浮動小数点数の組み合わせは加算を行う．文字列同士，リスト同士は連結を行う：

```
print(3+4.5) # 7.5
print("tulip"+"cosmos"+"pansy") # tulipcosmospansy
print([[1,2],"tulip"]+[3]) # [ [ 1, 2], "tulip", 3]
```

4-10 -（減算演算子）

（左オペランド，右オペランド）の組み合わせは，(整数，整数)，(整数，浮動小数点数)，(浮動小数点数，整数)，(浮動小数点数，浮動小数点数)．

減算を行う.

4-11 <<（左シフト演算子）

（左オペランド，右オペランド）の組み合わせは，（整数，整数）.
算術左シフトを行う.

4-12 >>（右シフト演算子）

（左オペランド，右オペランド）の組み合わせは，（整数，整数）.
算術右シフトを行う.

4-13 &（ビット論理積演算子）

（左オペランド，右オペランド）の組み合わせは，（整数，整数）.
ビット論理積を行う.

4-14 ^（ビット排他的論理和演算子）

（左オペランド，右オペランド）の組み合わせは，（整数，整数）.
ビット排他的論理和を行う.

4-15 |（ビット論理和演算子）

（左オペランド，右オペランド）の組み合わせは，（整数，整数）.
ビット論理和を行う.

4-16 &&（論理積演算子）

（左オペランド，右オペランド）の組み合わせは任意である.
まず左オペランドを評価し，それが偽ならば右オペランドの評価を行わず，偽を返す．左オペランドが真ならば，右オペランドの評価を行いその結果を返す．すなわち，ショートカット演算子として使用できる：

```
list = [1,2,3]
size(list) >= 1 && pop(list) # リスト内に要素が存在すれば pop する
```

4-17 ||（論理和演算子）

（左オペランド，右オペランド）の組み合わせは任意である.
まず左オペランドを評価し，それが真ならば右オペランドの評価を行わず，真を返す．左オペ

ランドが偽ならば，右オペランドの評価を行いその結果を返す．すなわち，ショートカット演算子として使用できる：

```
list = [1,2,3]
size(list) == 0 || pop(list) # リスト内に要素が存在すれば pop する
```

4-18 ..（範囲演算子）

（左オペランド，右オペランド）の組み合わせは，（整数，整数）.

左オペランドと右オペランドの大（小）に応じて，左オペランドから右オペランドまで 1（または-1）ずつ変化させた整数の集合を生成する．この集合は，左オペランドの整数と右オペランドの整数も含む．返されるのは整数の集合であり，リストではない：

```
print(0..5) # 0 1 2 3 4 5
print([2..-2]) # [ 2, 1, 0, -1, -2]
```

範囲演算子は for 文やリストスライスなどで有効に利用できる：

```
x = 0
for i in [0..10]
   x += i
print(x) # 55
list = [0,1,2,3,4,5,6,7]
print(list[2..4],[list[-1..-3]]) # 2 3 4 [ 7, 6, 5]
string = "tulip"
print(string[1..3]) # uli
```

4-19 ? :（三項演算子）

第一オペランド，第二オペランド，第三オペランドは任意の式.

第一オペランドが真の場合には第二オペランドが実行され，第一オペランドが偽の場合には第三オペランドが実行される：

```
def max(m,n)
   return m > n ? m : n

print(max(10,5)) # 10
```

三項演算子をネストすることで，多方向分岐を実現できる：

```
width = 35
size = width < 10 ? "small" : width < 20
     ? "medium" : width < 50 ? "large" : "huge!"
print(size) # medium
```

4-20 <，>，<=，>=（大小演算子）

（左オペランド，右オペランド）の組み合わせは，（整数，整数），（整数，浮動小数点数），（浮動小数点数，整数），（浮動小数点数，浮動小数点数），（文字列，文字列），（リスト，リスト）．

整数と浮動小数点数の組み合わせでは，通常の大小比較を行う．文字列同士の場合には，ASCIIコード順に基づいて大小比較を行う．リスト同士の場合には，リストの各要素について再帰的に大小比較を行った結果を返す：

```
print(3 < 4.5) # 1
print("tulipcosmos" >= "tulip") # 1
print([1,"tulip"] <= [1,"cosmos"]) # 0
print([1,2,3] < [[1,2],3])
   # エラー！（1 と [1,2] は型が違うため比較できない）
```

4-21 ==，!=（等号演算子）

（左オペランド，右オペランド）の組み合わせは，（整数，整数），（整数，浮動小数点数），（浮動小数点数，整数），（浮動小数点数，浮動小数点数），（文字列，文字列），（リスト，リスト），（ハッシュ，ハッシュ），（関数，関数），（クラス，クラス），（オブジェクト，オブジェクト），（null,null）．

==に関しては次の動作を行う．整数と浮動小数点数の組み合わせでは，数値が等しいときに真を返す．文字列同士では，文字の列として同じであれば真を返す．リスト同士では，同じリストを参照していれば真を返す．ハッシュ同士では，同じハッシュを参照していれば真を返す．関数同士では，同じ場所で定義された関数であれば真を返す．クラス同士では，同じ場所で定義されたクラスであれば真を返す．オブジェクト同士では，同じオブジェクトを参照していれば真を返す．null 同士では，真を返す．

!=は==の結果を否定した値を返す．

以下に例を示す：

```
print("tulip" == "tulip") # 1
list1 = [1,2,3]
list2 = [1,2,3]
print(list1 == list2) # 0
print({1->"one",2->"two"} == {1->"one",2->"two"}) # 0
tulip1 = new Tulip()
tulip2 = tulip1
print(tulip1 == tulip2) # 1
```

4-22 in，not in（包含演算子）

左オペランドは任意で，右オペランドはリスト．

in は，左オペランドを，右オペランドで指定されたリスト内の要素と 1 つずつ==演算子で比較していき，リストの中に一致する要素があれば真を，なければ偽を返す．

not in は in の結果を否定した値を返す：

```
point = 78
if point in [0..49]
   print("your grade is D")
elif point in [50..64]
   print("your grade is C")
elif point in [65..79]
   print("your grade is B")
else
   print("your grade is A")
```

5 文

5-1 規則

5-1-1 インデントルール

文構造はインデントによって管理される．プログラム全体に渡って，インデントをタブで行うか，あるいは半角空白で行うのかを統一する必要がある．

5-1-2 文の独立

文は単体で書かなければならない．すなわち，1 つの文の中に別の文を組み込むことはできない：

```
tulip = (cosmos = 1)+1    # エラー
list[++index] = 5    # エラー
```

5-1-3 コメント

#以降に記述された部分はコメントとして無視される．ブロック型のコメントは今はサポートしていない．

5-2 代入文

代入文は，変数に値を代入する．=を複数つなげることで，同じ値を複数の変数に代入することができる：

```
tulip = cosmos = pansy = "flower"
```

リスト代入を行うことができる：

```
list = ["tulip","cosmos","pansy"]
[x,y,z] = list
print(x,y,z) # tulip cosmos pansy
```

リスト代入を行う場合，右辺はリストである必要があり，その大きさが左辺の変数の数と一致していないとエラーになる．

また，代入文の右辺に現れる変数はすでにデータを保持している変数でなければならない．

5-3 累算代入文

+=, -=, *=, /=, %=, **=, <<=, >>=, &=, ^=, |=を使用できる：

```
tulip = 1
tulip += 2
print(tulip) # 3
```

累算代入文の左辺に現れる変数はすでにデータを保持している変数でなければならない．

5-4 インクリメント・デクリメント文

後置インクリメント，後置デクリメントを利用できる．前置インクリメント，前置デクリメントは許されない：

```
tulip++
++tulip # これはエラー
```

5-5 if 文

if 文は次の構造を取る：

```
if 条件部
    ... # 処理
elif 条件部
    ... # 処理
elif 条件部
    ... # 処理
else 条件部
    ... # 処理
```

elif ブロックはいくつあってもよい．else ブロックはあってもなくてもよい．

5-6 while 文

while 文は次の構造を取る：

```
while 条件部
```

```
    ... # 処理
```

条件部が真である限り，ブロック内の処理をループする．

5-7 for 文

for 文は次の構造を取る：

```
for 変数 in リスト
    ... # 処理
```

ループをまわる度に，変数がリストの次の要素を順に代入していく．
具体例を示す：

```
sum = 0
for i in [0..5]
   sum += i
print(sum) # 15
```

5-8 break 文

break 文は，while ブロック内または for ブロック内でのみ使用することができる．break 文が実行された時点で，そのループを抜ける：

```
for i in [0..10]
   if i >= 5
      break
print(i) # 5
```

また，break 文をコンマ区切りで並べて書くと，break を書いた数だけ，連続してブロックを抜けることができる：

```
i = 0
while i < 2
   for j in [0..2]
      if i > j
         break,break
      else
         print("({i},{j})")
   # この文は，(0,0) (0,1) (0,2) のみを出力する
```

5-9 continue 文

continue 文は，while ブロック内または for ブロック内でのみ使用することができる．continue 文が実行された時点で，そのループ処理を中断し，次のループ処理へと移る：

```
for i in [0..10]
   if i >= 4
      continue
   print(i) # この文は，0 1 2 3 のみを出力する
```

5-10 pass 文

pass 文は，そのブロックの処理内容が存在しないことを表す．pass 文は見た目のわかりやすさのために書くものであり，書いても書かなくても変わらない：

```
def func(n)
   pass

i = 0
if i < 0
   pass
elif i == 0
    # pass 文を記述しなくてもよいが，インデントは必要である
else
   print("i is a positive integer")
```

プログラムの開発段階で，ひとまず関数名だけ書いておきたい，などの場合に役立つ．

5-11 select 文

select 文は，C 言語や Java の switch 文に相当する文である．次の構造を取る：

```
select 評価対象の式
   case 値
      ... # 処理
   case 値
      ... # 処理
   else
      ... # 処理
```

case 文はいくつあってもよい．else 文はあってもなくてもよい．case 文から抜けるには break 文を使う：

```
num = 8
if num > 10
   print("num must be in [1..10]")
else
   select num
      case 2
         pass
      case 3
```

```
            pass
        case 5
            pass
        case 7
            print("{num} is a prime number.")
            break
        else
            print("{num} is not a prime number.")
            # 8 is not a prime number. と表示される.
```

5-12 def 文，return 文

関数は次のようにして定義する．return 文によって戻り値を返すことができる．コンマで区切ることで，複数の戻り値を返せる：

```
def func(i,j)
    if i < j
        return "{j} is larger than {i}. abs({i}-{j}) = ",j-i
    elif i > j
        return "{i} is larger than {j}. abs({i}-{j}) = ",i-j
    else
        return "{i} and {j} is equal."

print(func(5,3)) # 5 is larger than 3. abs(5-3) = 2
print(func(7,10)) # 10 is larger than 7. abs(7-10) = 3
```

引数の個数が違う関数をオーバーロードすることができる：

```
def func(i)
    ... # 内容

def func(i,j)
    ... # 内容

def func(i,j,k)
    ... # 内容
```

関数は文を記述することができる任意の場所に記述可能だが，あとで述べるスコープの概念の関係上，関数の中に関数をネストすることは推奨されない．

5-13 try 文，throw 文

%%% 詳細な仕様は未定．モジュールとして実装する．

try 文は例外処理を実現する；

```
try
```

```
      tulip = cosmos/pansy
   catch INCORRECT_TYPE_EXCEPTION
      print("cosmos and pansy must be integer or float.")
   catch ZERO_DIVISION_EXCEPTION
      print("division by 0 occured.")
   else
      print("some exception occured.")
      throw
```

try 文によるブロック内部の処理が例外捕獲の対象になる．このブロック内で例外が捕獲された場合，この例外があとに続く catch 文から検索される．

該当の catch 文が見つかると，その catch 文のブロックの内容が実行された後，プログラムが終了する．ただし，catch 文の内部に throw 文が記述されていると，現在処理している例外が，現在処理している関数を呼び出した関数へと送出される．

該当の catch 文が見つからない場合で，かつ else 文が記述されている場合には，else 文のブロックの内容が実行された後，プログラムが終了する．この場合にも，else 文の内部に throw 文が記述されていると，現在処理している例外が，現在処理している関数を呼び出した関数へと送出される．else 文が記述されていない場合には，該当の例外を表示してプログラムが終了する．

また，例外がグローバル領域に達した場合にも，該当の例外を表示してプログラムを終了する．

else 文はすべての例外を吸収してしまうため容易な使用はあまり勧められない．

5-14 クラスとオブジェクト

クラスを定義し，そのインスタンスとしてオブジェクトを作成することができる．

クラスの構造は次の通りである：

```
class クラス名
   クラス変数の初期化など

   def new（引数リスト）   # コンストラクタ
      インスタンス変数の初期化など

   def 関数1（引数リスト）
      ... # 内容

   def 関数2（引数リスト）
      ... # 内容
```

クラスの最初では，クラス変数の初期化を行う．ここで定義されていないクラス変数は用いることができない．クラス変数の初期化は，クラスブロックにおけるトップレベルなら任意の場所

に記述可能だが，最初に書いておくのが好ましい．

コンストラクタは，new という名前の関数として定義する．コンスラクタ内では，使用するインスタンス変数全ての初期化を行わなければならない．ここで初期化されていないインスタンス変数は使用することができない．コンストラクタはオーバーロード可能である．

クラスには関数を複数定義することが可能である．クラスを記述する際に，静的関数（クラス関数）とインスタンス関数の区別はない．関数はオーバーロード可能である．

クラスの中で，クラス変数にアクセスするには次のようにする：

クラス名．クラス変数名

クラスの中で，インスタンス変数にアクセスするには次のようにする：

this．**インスタンス変数名**

ただし，ローカル変数にインスタンス変数名と同名の変数が存在しない場合には，this．を省くことができる．これに関してはスコープのところで詳しく述べる．

次にオブジェクトについて説明する．クラスのオブジェクトは次のようにして作る：

オブジェクト = new **クラス名（引数リスト）**

引数リストの引数の数に応じて，適切なコンストラクタが呼び出される．該当するコンストラクタが存在しない場合にはエラーになる．

インスタンス変数にアクセスするには次のようにする：

オブジェクト．インスタンス変数名

インスタンス関数を呼び出すには次のようにする：

オブジェクト．関数名（引数リスト）

クラス変数にアクセスするには次のようにする：

クラス名．クラス変数名

静的関数を呼び出すためには次のようにする：

クラス名．関数名（引数リスト）

すなわち，クラス定義中の関数には静的関数とインスタンス関数の違いは存在しないが，関数を呼び出すときに，それが静的関数であるかインスタンス関数であるかが決定される．ただし，静的関数として呼び出した関数が，インスタンス変数へのアクセスを伴うような関数の場合にはエラーになる．

クラスとオブジェクトについて具体的な例を示す：

```
class Tulip
   number = 0 # クラス変数
```

```
    def new(color)  # コンストラクタ
       this.color = color # インスタンス変数を初期化
       ++Tulip.number # Tulip の本数を増やす

    def set_color(color)
       this.color = color # 色を設定する

    def get_color()
       return color # 色を返す

    def get_flower_name()
       return "Tulip" # 花の名前を返す

tulip1 = new Tulip("blue") # Tulip クラスのオブジェクトを作成
print(tulip1.get_color(),tulip1.color,Tulip.number)
   # blue blue 1
tulip2 = new Tulip("yellow") # Tulip クラスのオブジェクトを作成
print(tulip2.get_color(),tulip2.color,Tulip.number)
   # yellow yellow 2
print(Tulip.get_flower_name(),tulip1.get_flower_name())
   # Tulip Tulip
```

クラスは名前空間を提供すると考えることができる：

```
class Math
   pi = 3.1415926535
   e = 2.7182818284

   def new()  pass

   def area(radius)
      return radius*radius*Math.pi

   r = 1
   print(Math.area(r)) # 3.14
   print(Math.e) # 2.7182818284
```

5-15 イベントリスナ

%%% 詳細な実装は未定．モジュールとして整える予定だが，そもそも実装するかどうかも微妙.

関数をイベントリスナに登録することで，イベントドリブンなプログラミングモデルを実現することができる：

```
listener SocketListener
   case TIME_OUT
      ... # 接続がタイムアウトした場合の処理を記述
   case SOCKET_CLOSED
      ... # 接続先のソケットが閉じられた場合の処理を記述
   default
      ... # その他の場合の処理を記述

socket_listener = new SocketListener() # リスナオブジェクトを作成
handle = socket_listener.add(send,from_addr,to_addr,message)
   # send 関数をリスナに登録し，ハンドルを取得
socket_listener.remove(handle)
   # handle で指定される関数をリスナから強制的に回収
```

5-16 use 文とモジュール組み込み

use 文を記述することにより，モジュールを組み込むことができる．

モジュールとは，クラスをいくつか記述したファイルである．たとえば次のプログラムを flower.iris として保存する：

```
class Tulip
   ... # クラスの内容を記述

class Cosmos
   ... # クラスの内容を記述
```

use 文を利用すると，それを記述した位置に指定したファイルが読み込まれる．ファイル名は¨と¨で囲む必要がある：

```
use "flower.iris" # モジュール組み込み
tulip = new Tulip() # Tulip クラスのオブジェクトを作れる
cosmos = new Cosmos() # Cosmos クラスのオブジェクトを作れる
```

モジュールは常にクラスの集まりとして提供される．モジュールは，プログラムの再利用性や拡張性を確保する．

6 スコープと変数

6-1 変数の種類

Iris に存在する変数には 3 種類あり，それはローカル変数とクラス変数とインスタンス変数である．グローバル変数は存在しない．

6-2 Main クラス

プログラムを開始したとき最初に実行されるのは，プログラムのトップレベルに直接記述された部分である．この部分はどのクラスにも属していないが，これは Main クラスに属していると見なす．たとえば，次のようなコードを考えてみる：

```
def is_prime(n)
   for i in [2..n-1]
      if !(n%i)
         print("{n} is not a prime number!")
         return
   print("{n} is a prime number!")
   return

i = 2
while i < 100
   is_prime(i)
```

ここで，上のコードは Main クラスに属していると考える．つまり上のコードは次のコードと等価であると考える：

```
class Main
   # Main クラスにはクラス変数は存在しない

   def new()  # コンストラクタ
      i = 2
      while i < 100
         is_prime(i)

   def is_prime(n)
      for i in [2..n-1]
         if !(n%i)
            print("{n} is not a prime number!")
            return
      print("{n} is a prime number!")
      return
```

そして，プログラムが実行されるとは，Main クラスのオブジェクトが 1 つだけ作成されることだと考えればよい．このような見方をすれば，プログラムのトップレベルに書かれたコードも，クラスであると見なせるので，統一的な理解が可能となる．また，グローバル変数という概念が必要なくなる．

6-3 スコープ

スコープは変数の有効範囲を作り出す．スコープ内で作成された変数は，スコープ内でのみ寿命を持ち，スコープ外では無効である．Iris では，スコープを作り出すのはクラスと関数のみである．クラスはクラス変数とインスタンス変数のスコープを作り出し，関数はローカル変数のスコープを作り出す．つまり，Perl などと違って，`if` 文や `for` 文のブロックがスコープを作り出すことはない：

```
def func(m,n)
   if m < n
      flag = 1
   else
      flag = 0
   print(flag) # flag にアクセス可能

func(1,2)
flag = 3 # エラー！
```

6-4 クラス変数，インスタンス変数，ローカル変数

クラス変数にアクセスするには，それがクラスの内部であっても外部であっても，次のように記述すればアクセス可能である：

クラス名 . クラス変数名

ローカル変数にアクセスするには，特に修飾することなく，次のように記述する．ローカル変数にはそれが属する関数の内部以外からはアクセスできない：

ローカル変数名

インスタンス変数にアクセスするには，それがクラスの外部からのアクセスである場合には次のように記述する：

オブジェクト名 . インスタンス変数名

クラスの内部からアクセスするには次のように記述する：

`this` **. インスタンス変数名**

ただし，this．を付けずに変数名へのアクセスが行われ，かつその名前のローカル変数が存在しない場合には，インスタンス変数が参照される．すなわち，同名のローカル変数が存在しない限り，this．を付ける必要はない．

6-5 Main クラスにおけるクラス変数，インスタンス変数，ローカル変数

Main クラスにはクラス変数は存在しない．

ローカル変数は何も付けずにその変数名を書けば良い．

C 言語や Perl などで「グローバル変数」と呼ばれるものにアクセスするには，this．を付ける．もちろん，同名のローカル変数が存在しない場合には，this．は付けなくても良い：

```
count = 0
for i in [0..100]
   if i%3
      this.count++ # this. は省略可能
print(count) # 34
```

上記のコードを Main クラスに入れてみると，次のようになる：

```
class Main

   def new()
     count = 0
     for i in [0..100]
     if i%3
        this.count++ # this. は省略可能
      print(count) # 67
```

実行されるプログラム本体が Main クラスから生成されるオブジェクトであるので，「グローバル変数」に this．を付けてアクセスするのは自然だろう．

6-6 具体例

以上で述べたことを具体的なプログラムで説明する：

```
class SomeClass
   var = 0

   def new(var)
      this.var = var

   def print(var)
      print(var,this.var)

def some_func(var)
```

```
    object = new SomeClass(var)
    object.print(var)
    object.print(this.var)

var = 0
some_func(1) # 1 1 0 1 と表示される
```

7 組み込み関数

%%% 原則的に「Perl で頻繁に使うもの」を用意する.

7-1 基本的な関数

7-2 文字列を取り扱う関数

7-3 リストを扱う関数

7-4 ハッシュを扱う関数

7-5 数学関数

7-6 システム関数

8 組み込みモジュール

%%% 詳細は未定だが，例外処理，スレッド，ファイル I/O は必須．ソケットもぜひ実装する．イベントリスナは実装するかどうかは未定．GTK モジュールは時間があれば実装する．

8-1 ファイル I/O モジュール

8-2 ソケットモジュール

8-3 例外処理モジュール

8-4 スレッドモジュール

8-5 イベントリスナモジュール

8-6 GTK モジュール