B演習 (言語処理系演習) 資料 (6)
評価器

田浦

2006/12/11

1 概要

mini-Python の式を評価した結果として現れるあらゆるデータが，処理系 (C のプログラム) 内でどう表現されているか (データ表現) が決まると，いよいよ我々の最終目的である，式，文，それらの並び，ひいてはプログラム全体の評価器を作成することができる．式の評価器 (`eval_expr`) とはつまり，式を与えられると，それを評価した値をデータ表現にしたがって作り出す C 関数のことである．文の評価器 (`eval_stmt`) とはつまり，文を与えられると，その文を順次実行したときに生じる副作用を生じさせる (例えば print 文の実行にともなって値を実際に出力したり，代入文の実行に伴って実際に変数の値を書き換えたりする) とともに，次に実行すべき文の選択に必要な情報を値として返す．プログラム全体を評価するとはつまり，プログラムに並んでいる文 (大きなプログラムでは，その多くは `def` 文であろう) を順に実行するだけのものである．

それぞれの評価器はこのほかに，実行時エラーが生じた際に適切なメッセージおよびエラーが生じた場所 (ソースファイル上の位置とその時点での関数呼び出し履歴，いわゆるスタックトレース) に関する情報を表示する役割も持つ．正しいプログラムを正しく実行することのほかに，エラーを生じたプログラムに対して，適切な情報を表示するのも，言語処理系の重要な役割である．

さて，我々はすでにすべての mini-Python に現れるデータの表現として，形式的には `py_val_t` という型 (`py_val` という構造体へのポインタ) を使うことに決めているので，評価器を作るとは，少なくとも形式的には以下の 3 つの関数を完成させることである．

- `py_val_t eval_expr(expr_t` e`, ···)`
- `py_val_t eval_stmt(stmt_t` s`, ···)`
- `py_val_t eval_file_input(file_input_t` u`, ···)`

··· 部分には，エラーを表示するために必要な引数をはじめ，後に説明する引数が入る．

これらの関数ができれば，我々のインタプリタ本体は，以下のようにしてほぼ完成を見ることになる．

```
/* filename からプログラムを読み込み実行 */
int exec_file(char * filename, ..., ...)
```

```
{
  tokenizer_t t = mk_tokenizer(filename);
  file_input_t u = parse_file_input(t);
  eval_file_input(u, ...);
  return 0;
}
```

ここでも，··· 部には後に説明する引数が入る．

eval_expr や eval_stmt は構文木を見て，式や形の種類に応じて適切な評価を行うので，実際にはこれらを作るために，いくつもの下請け関数を作ることになる．

2 評価のための重要な概念: 環境

2.1 概念

文や式を評価する際の重要な概念である**「環境」(environment)** というものについて説明する．環境とは，早い話が変数とその値とを保持している写像のことである．例えば我々は，

```
x = 1
print x + 5
```

というプログラムが何を表示するかを考える際に,「式 x + 5 を評価する際に x の値が 1 だから，全体としてこの値は 6」という思考をしている．この,「x の値が 1」ということを保持しているものが環境である．

また，環境と言う考え方は，局所変数 (またはローカル変数; 関数内でのみ有効な変数) というものの挙動を説明するのにも役に立つ．例えば，

```
def f():
  x = 1
  return x

def g():
  x = 2
  f()
  return x
```

とすると，当然ながら g() は 2 を返す．ポイントは，x = 2 のあと f() によって x = 1 という代入がなされているが，これが g 内で定義された x を書き換えるわけではないことである．

環境と言う概念を用いこれを説明すると，f() という関数呼び出しが用いる環境と g() という関数呼び出しが用いる環境が異なる，という説明が可能になる．ローカル変数はおよそどんなプログラム言語でも備えているから，それを環境と言う概念で定義・説明するのは，プログラム言語の意味 (仕様) の定義の仕方としてかなり一般的な方法である．

この言葉を使って言えば，関数呼び出しは「**新しい環境を作り**，その環境に引数の値をセットし，その環境の下で関数本体を評価する」という行いであると説明できる．**代入文は，環境を書き換える操作**である．

さらに，若干 Python 固有の話になるが，大域変数と局所変数の違いも以下のように説明できる．復習をしておくと，Python には大域変数と局所変数の 2 種類があり，以下の規則がある．

- 代入文「$x =$ 式」は，
  - それがプログラムのトップレベル (どの関数定義内でもない場所) に書かれえいる場合，大域変数 x に値を代入する．
  - 関数の定義 (`def` 文) 内に書かれている場合，その呼び出し内にのみ有効な局所変数 x に値を代入する．
  - ただし例外があり，その呼び出し中に `global` x が実行されていた場合，その呼び出し中は x を常に大域変数とみなす．
- 変数 x への参照は，
  - それがトップレベルに書かれた式であった場合，大域変数 x を参照する．
  - 関数定義内に書かれた式であった場合，もしその呼び出し中に，すでに局所変数 x への代入が行われている場合，局所変数 x を参照する．そうでなければ大域変数 x を参照する．

同じことを，環境と言う言葉を用いて明確にすると以下のようになる．まず，

- プログラム実行中唯一の**大域環境 (global environment)** がある．
- 各関数呼び出しの際に新しく環境が作られ，その呼び出しの評価のために使われる．この環境を，その呼び出しのための**局所環境 (local environment)** と呼ぶことにする．**便宜上，プログラムのトップレベルに書かれた式は，局所環境として，上記の大域環境を用いて評価されていると考える．**

その上で，

- `global` x は，現在の局所環境に，x が大域変数であることを示す特別な印をつけておく
- 代入文「$x =$ 式」は現在の局所環境に，x が大域変数であることを示す印がついていれば大域環境の，そうでなければ局所環境の x に値を代入する
- 変数 x への参照は，局所環境に x が大域変数であることを示す印がついていれば，大域環境の，x を参照する．そうでなければ，局所環境の x を参照する．後者の場合，見つからなければ続けて大域環境の x を参照する．それでも見つからなければ実行時エラーとなり，エラーメッセージを表示してプログラムを終了する．

このように，環境と言うものを導入したおかげで「変数の値を決める規則」が明確化された．

環境は，プログラマや言語仕様の設計者にとっては，プログラムの挙動を説明するための一種の説明方法に過ぎないが，処理系を作成する際にはそれを明示的にデータとして，処理系内部で構築

することになる．mini-Python の場合，局所関数と大域環境という二つの環境が実際に処理系内部でデータとして作られ，eval_expr, eval_stmt は，局所環境と大域環境を引数として明示的に受け取る関数となる．プログラム全体を評価する関数 eval_file_input は，局所変数を受け取る必要はなく，大域環境だけを受け取る．以下では，環境をあらわすデータ構造 (へのポインタ)env_t という型が定義されていると仮定して，環境というデータ構造のインタフェースを示す．環境のほかに，実行時エラーの際にその時点の関数呼び出しスタックを保持するデータ (stack_trace_t 型のデータ) を，さらなる引数として渡している．まとめると，インタフェースは以下のようになる．

- py_val_t eval_expr(expr_t e, env_t $lenv$, env_t gnv, stack_trace_t bt)

  式 e を局所環境 $lenv$, 大域環境 $genv$ の元で評価した結果を返す．bt はその式を評価する時点での関数呼び出し履歴を保持しており，もし e を評価している途中で実行時エラーが生じたら，その時点での関数呼び出しを表示してプログラムを終了する．

- py_val_t eval_stmt(stmt_t s, env_t $lenv$, env_t gnv, stack_trace_t bt)

  文 s を局所環境 $lenv$, 大域環境 $genv$ の元で評価した結果を返す．bt はその文を評価する時点での関数呼び出し履歴を保持しており，もし s を評価している途中で実行時エラーが生じたら，その時点での関数呼び出しを表示してプログラムを終了する．

- void eval_file_input(file_input_t u, env_t $genv$, stack_trace_t bt)

  ファイルに書かれているプログラム (をあらわす構文木)u を大域環境 $genv$ の元で評価する．bt はそのプログラムを評価する時点での関数呼び出し履歴を保持しており，もし u を評価している途中で実行時エラーが生じたら，その時点での関数呼び出しを表示してプログラムを終了する．

env_t, stack_trace_t という部品の実装方法は後に説明する．$lenv$ は local environment の略で，文字通り局所環境と言うことである．$genv$ は global environment の略で，大域環境と言うことである．bt は backtrace の略である．

2.2 環境の外部インタフェース

以上の概念を理解すれば，以下はほとんど説明が要らないはずである．

- mk_env():

  空の環境を作る．

- void env_set(env_t env, char * s, py_val_t v):

  変数名 s と値 v を与え，環境 env に $s \to v$ という写像 (変数 s の値が v である情報) を登録する．

- py_val_t env_lookup(env_t env, char * s):

  変数名 s を与えて，それに対応する値 v を取り出す．ただし，s が環境に登録されていない場合，そのことを示す特別な返り値 (py_val_not_found) を返す．

- `void env_set_global(env_t` *env*`, char *` *s*`)`:

  変数名 *s* を与え，環境中で *s* が大域変数であることを登録する．

- `py_val_t env_is_global(env_t` *env*`, char *` *s*`)`:

  変数名 *s* を与え，環境中で *s* が大域変数であるかどうかを返す．

2.3 環境の実装

環境の実装は非常に基本的なデータ構造の練習問題である．しかも，env_lookup は変数を参照するたびに呼ばれる関数なので，処理系全体の速度に影響が大きい部分である．どのようなデータ構造を用いればよいか，またどうすれば速くできるかを考えてほしい (もちろん，最初は「動く」ことを優先して単純なつくりにして置けばよい)．

例えば，の話であるが，一番簡単には，以下のような,「変数名とその値」を組とした構造体 (エントリ) を作り，環境はその配列 (後から値を追加できる必要があるので，我々が各所で作っているベクタ) として作ることができる．env_lookup はそのベクタを線形探索する．env_set はそのベクタを変数名で検索した上で，見つかればそのエントリの val フィールドを書き換え，なければ新しいエントリをベクタに追加すればよい．注意としては，探索をする際に二つの文字列を==で比較しないことである (なぜか? C 言語の基本的な落とし穴)．

```
typedef struct env_entry
{
  char * name;
  py_val_t val;
} env_entry, * env_entry_t;
```

それ以上は各自考えてほしい．ここで示したのは非常に単純な一方法に過ぎない．

環境中に変数名が登録されていなかった場合には，それを示すある特別な値が返される．その「特別な値」はすでに pyvalues.h に，py_val_not_found というシンボルとして定義されており，先週のデータ表現において用いられたどのような値ともぶつからないようなビット列を利用する．

また，env_set_global, env_is_global は，それぞれ env_set, env_lookup と似た処理をすることになることが想像できるだろう．実際最も簡単な方法は，py_val_t 型のデータのひとつとして，「変数が大域変数である」ということを示す特別な値を追加することである．環境を探索して返された値がその特別な値であれば，その変数が大域変数であることを示す．env_set_global(*env*, *s*) は，*s* の値としてその特別な値を登録するだけである．

この値も pyvalues.h に，py_val_global というシンボルとして，上記の py_val_not_found の定義の周辺で，定義されている．このもとでは，env_set_global と env_is_global は以下でよい．

```
int env_is_global(env_t env, char * key)
{
  return env_lookup(env, key) == py_val_global;
}
```

```
void env_set_global(env_t env, char * key)
{
  env_set(env, key, py_val_global);
}
```

残る mk_env, env_set, env_lookup は各自実装すること.

3 スタックトレース (関数呼び出し履歴) を保持するデータ構造

注: 本節は実行時エラー発生時に親切なエラーメッセージを表示するための仕組みについての説明である. 最初は読み飛ばしてもよい.

3.1 実行時エラーメッセージの表示

mini-Python プログラム実行中には様々なエラーが生ずる. それらのエラーがプログラム実行中に生じた場合, 有用なエラーメッセージを表示してプログラムを終了させる必要がある. 有用なエラーメッセージにはエラーが生じた場所 (ソース上の位置) の他に, そこにいたるまでの**関数呼び出し履歴**がある.

たとえば以下のような間違ったプログラム (h 内の a[0] において a が文字列, タプル, リスト, 辞書のいずれでもない) を実行した際に,

```
def f(a):
    return g(a)

def g(a):
    return h(a)

def h(a):
    return a[0]

f(3)
```

以下のようなエラーメッセージを表示するのはそこそこ親切といえよう (もちろん各自の処理系で, エラーメッセージの詳細は異なっていて良い).

```
enshu:pi% ./minipy a.py
_builtin.py:133 runtime error:
    type error (non-subscriptable object)
stack trace:
a.py:10 f
a.py:2 g
```

```
a.py:5 h
a.py:8 getitem
```

1行目は最終的にエラーが起きたソースの場所 (`_builtin.py` の 133 行目) を示している．なぜ，`a.py` というプログラムを実行したにもかかわらずこのような見覚えがないファイル名が表示されるかについては後に述べる．ここでは，処理系内部のライブラリに実行が到達したと思ってもらえばよい

その次の行でエラーの内容 (種類) に関する簡単な記述がくる．ここでは，”type error (non-subscriptable object)”と記述されている．

そのあと (`stack trace:`以下の行) に，現在の場所に至った関数呼び出しの履歴 (スタックトレース) が表示されている．各行は，関数呼び出しが起きた地点と，そこで呼ばれた関数名を表示している．一番上の上の行 (`a.py:10 f`) がトップレベルから呼ばれた関数で，その呼び出しが `a.py` の 10 行目でおきたことを示している．その下が `f` から呼び出された関数であり，`g` が `a.py` の 2 行目で呼びされたことを示している．

このような情報を表示しておけば，プログラマが mini-Python プログラムをデバッグするのに役に立つであろうし，処理系を作っている最中においては，処理系自身の間違った挙動を突き止めるのにも役に立つ．

このような情報をきちんと表示するために処理系内部で維持されていなくてはならない情報は，

各式や文について，そのソース上での位置: つまり各式や文が書かれているファイル名と行番号のことである．もしその式や文を評価している最中にエラーが起きた場合，それをエラーの場所として表示する．また，下で述べる，関数呼び出しの履歴を維持するためにも使う．

関数呼び出しの履歴: トップレベルからエラー発生地点に至るまでの関数呼び出しの列で，いわゆるスタックトレースである．これは文字通り，呼び出された関数名とその呼び出しがおきたソース上での位置をスタック状に保持しているデータ構造であり，評価器が関数呼び出しを実行する際にその呼び出しに関するエントリを追加 (push) し，その呼び出しがリターンしたときにそのエントリを削除 (pop) することで維持される．

前者については，構文解析器が構文木データの中にすでに埋め込んでいる (`expr_t` や `stmt_t` 中の `src_pos_t pos` というフィールドがそれである)．

ここでは後者を保持するデータ構造のインタフェースとその実装について述べる．といっても，説明はほとんど上記で尽きている．

3.2 スタックトレースの外部インタフェース

- `stack_trace_t mk_stack_trace()`

  空のスタックトレースを作る．

- `void stack_trace_push(stack_trace_t` *bt*`, char *` *name*`, src_pos_t` *call_site*`)`

  スタックトレース *bt* に,「*name* とい関数がソース位置 *call_site* で呼ばれた」というエントリを記録 (push)．

- `void stack_trace_pop(stack_trace_t` *bt*`, char *` *name*`, src_pos_t` *call_site*`)`

  スタックトレース *bt* から頂上のエントリをひとつ除去 (pop) する．それは,「*name* とい関数がソース位置 *call_site* で呼ばれた」というエントリでなくてはならず，そうでなかった場合はエラー (評価器の実装が間違っている) である．*name*, *call_site* は評価器の実装のエラーを検出するためだけに渡される．

- `void print_stack_trace(stack_trace_t` *bt*`)`

  スタックトレース *bt* を標準エラーに表示する．実行時エラーが発生した際にこの関数が呼ばれる．

3.3 スタックトレースの実装

環境同様，特に難しいことはない単純な部品のはずである．自然には，まずひとつの関数呼び出しのエントリを記録する以下のような構造体を定義する．

```
typedef struct stack_trace_entry
{
  char * name;
  src_pos_t call_site;
} stack_trace_entry, * stack_trace_entry_t;
```

目的の `stack_trace_t` は，この `stack_trace_entry_t` を push/pop できるスタックとして実現すればよく，これまでに何箇所かで作ったはずの `XXX_vec` という部品がここでも少しの変更で再利用できるはずである．

各自実装すること．

4 式の評価

4.1 概要

さて，環境というお膳立てや，エラーメッセージの表示の仕方と言う若干のわき道が長くなってしまったが，いよいよ「式の評価器」の作成に取り掛かる．mini-Python には以下の種類の式が存在した．

- 変数 (`expr_kind_var`)
- リテラル (`expr_kind_literal_int`, `expr_kind_literal_float`, `expr_kind_literal_string`, `expr_kind_none`)
- タプル構築式 (`expr_kind_display_tuple`)，リスト構築式 (`expr_kind_display_list`)，辞書構築式 (`expr_kind_display_dict`)

- 括弧 (expr_kind_paren)
- 関数呼び出し (expr_kind_call)
- 演算子 (expr_kind_operator)
- 属性参照 (expr_kind_attref)
- 添え字 (expr_kind_subscript)

式の評価器はこれらの各場合について網羅的に,「それを評価した値のデータ表現—py_val_t 型のデータ」を作り出す関数 eval_expr である.

つまり, eval_expr の概形は以下のようになる.

```
py_val_t eval_expr(expr_t exp, env_t lenv, env_t genv, stack_trace_t bt)
{
  switch (exp->kind) {
  case expr_kind_var: /* variable */
    return ...
  case expr_kind_literal_int: /* int literal */
    return ...
  case expr_kind_literal_float: /* float literal */
    return ...
  case expr_kind_literal_string: /* string literal */
    return ...
  case expr_kind_none: /* None literal */
    return ...
  case expr_kind_display_tuple: /* [ a, b, c,...] */
    return ...
  case expr_kind_display_list: /* [ a, b, c,...] */
    return ...
  case expr_kind_display_dict: /* { a : x, b : y } */
    return ...
  case expr_kind_paren: /* ( e ) */
    return ...
  case expr_kind_operator: /* all kinds of operators */
    return ...
  case expr_kind_attref: /* e.f  => attref(e, "f")?? */
    return ...
  case expr_kind_subscript: /* e[e] => subscript(e, e) */
    return ...
  case expr_kind_call: /* e(e:e,..) */
    return ...
```

```
  default:
    bomb();
    return 0;
  }
}
```

それにしてもこう場合わけが多くてはやる気をそがれてしまうかもしれない．以下は勇気付けのための材料である:

- 最初の 5 つは，実際非常に簡単なはずである．データ表現に従ったデータを作り出せば良いだけの話である．
- expr_kind_paren の場合 (式を括弧でくくったもの) は，要するに括弧でくくられた式を評価してそれをそのまま返せばよいだけで，やはり簡単のはずである．
- タプル構築 (expr_kind_display_tuple), リスト構築 (expr_kind_display_list), 辞書構築 (expr_kind_display_dict) もさほど難しくない上に，評価器が行う仕事は 3 つの処理に対して大部分が共通である．

  つまり，要素の位置に書かれている式をことごとく評価し，あとは処理系内部で定義されているはずの，タプル，リスト，辞書を新たに作る関数を呼び出すだけである．もちろんそれらのデータ構造を作る部分は，別途作ってお子なくてはならないので，そこで多少苦労するかもしれないが.
- **自信のない人は，まず縮小仕様 (sub-Python) の処理系を作ると良い**．つまり，文字列，タプル，リスト，辞書などのややこしいデータ構造は，他ができて，それで動く範囲のプログラムが動くようになってから作ることにする．これにより，ひとまず整数や浮動小数点数のみを扱うプログラムが最短距離で実行できるようになる．もちろん文字列がなければ"hello world" すら表示できないが，ループや関数呼び出しを含んだ単純な数の計算は，行えるようになる．実装すべき項目が多くてめげそうな人はこの道をとることを勧める．そうでない人も，ひとまずそれらのデータのみを使う処理系を一度作ってから，他のデータ構造を後から追加するという方法をとっても良いかもしれない.

  以降では，sub-Python においては必要のない，あるいは単純化される項目に，(∗) の印をつけておいた.
- expr_kind_attref の場合は，4.6 節に述べる事情により，単にエラーにして置けばよい．我々は一般の属性参照式をサポートしているわけではないからである．
- 演算子 (expr_kind_operator), 添え字 (expr_kind_subscript), 関数呼び出し (expr_kind_call) は，文法は違うが，**評価の方法はほとんど共通**である．**実際前者二つも，関数呼び出しの特別な場合といってよい．**つまり評価器内部では，前者二つは，何か適当な組み込み関数の呼び出しとみなしてしまえばよいのである．たとえば,

$$E_1 + E_2$$

と言う式を評価器が評価することになったら，それを，

$$\texttt{add}(E_1, E_2)$$

という関数呼び出しとみなし，あとは関数呼び出しの場合と同じ処理をする，ということにしてしまう．添え字式:

$$E_1\texttt{[}E_2\texttt{]}$$

の場合も，たとえば，

$$\texttt{getitem}(E_1, E_2)$$

という呼び出しとみなしてしまう．これによって，この 3 つの場合で，大部分の処理を共通化できる．

ではそれらの add や getitem などの関数はどこでどう実現されているのか?

ある場合にはそれは，mini-Python の native 関数として処理系内に実装されている．処理系内に対応する関数をつくっておき，それをもとに，native 関数を処理系立ち上げ時に大域環境に登録しておくのである．そうすれば，通常の関数呼び出し評価の規則にしたがって大域環境が探索され，呼び出される．細かく言えば，通常の関数呼び出し評価の規則にしたがった場合，局所関数が最初に探索されるので，そこに同名の変数 (関数) が定義されていたら正しい呼び出しは行われない．我々はこれには眼をつぶる[1]

また別の場合 (こちらの方が一般的) には，それはある mini-Python プログラム中に通常の def 文を用いて定義されている (もちろんそれが途中で native 関数を呼び出すことはある)．そのプログラムは「組み込みライブラリ」として，処理系が定めた場所 (たとえ場処理系と同じディレクトリ内) に置かれ，すべてのプログラムの実行に先立ってこっそり実行される．概念的には，その組み込みライブラリとユーザが渡したプログラムを連結して実行しているに過ぎない．実は，3.1 節のエラーメッセージ中に登場した_builtin.py がそれである．

それでは個別のケースを見ていく．以下では，評価すべき式の構文木 (expr_t 型) を e と書く．

4.2 変数

1. 変数名 (e->u.var) を局所環境で探索 (env_lookup) し，値が見つかればそれを返す.
2. 見つからないか (py_val_not_found が返された)，または大域変数であることが示されていれば (py_val_global が返された)，大域環境を探索する.
3. 見つかればそれを返し，さもなければ実行時エラー (未定義変数の参照) を起こす.

[1]仕様上は．add，getitem などの名前を「予約された」関数名とし，予約された関数前を再定義してはいけないことにすればよい．しかし面倒なのでいちいち検査することはしなくてよいことにする.

4.3 リテラル

int, float, string の各場合について，値 (e->u.lit_i, e->u.lit_f, e->u.lit_s) に対応するデータ表現を作って返す．None の場合はさらに簡単である．それらを作るための関数は pyvalues.c に定義されている．

4.4 タプル，リスト，辞書構築式 ($*$)

自信のない人は，ひとまず後回しにしても良い．

準備として，py_val_t 型の値を要素として格納するベクタ py_val_vec_t を定義しておく．これまでに何度も作った，XXX_vec という部品の作り方を踏襲すればよい．

1. 空のベクタ (py_val_vec_t) を作成する．ここに，要素式を評価した結果を格納していく．
2. すべての要素式 (辞書の場合，キーと値をすべて; e->u.disp) を，再帰的に評価器を呼ぶことによって評価し，上で作ったベクタに加えていく．
3. できたベクタを，タプル，リスト，辞書を作る適切な関数 (mk_py_tuple, mk_py_list, mk_py_dict) に渡してデータを作り，それを返す．それらの関数は pyvalues.c 中で DO_IT_YOURSELF とマークされているもので，各自作ることを期待されている．

しかし，これがしんどいと思う人は，ひとまずここを飛ばすと良い．これらがないと，面白いプログラムの実行はできないが，整数や浮動小数点数だけをデータとするプログラムであれば実行できる．

4.5 括弧

括弧で囲まれた式に対して，再帰的に評価器を呼べばよい．

4.6 関数呼び出し

上で述べたとおり，関数呼び出し，演算子，添え字の 3 つの場合で，多くの共通部分があり，後ろの 2 者は，内部的には関数呼び出しとみなして処理する．

そのためここでは，関数呼び出しの説明を先にしておく．関数呼び出し式は一般には

$$E_0(E_1, \cdots, E_n)$$

の形をしている．ここで，各 E_i はやはり式である．この式を評価するとは，

1. 関数の位置にある式 E_0 を，評価器を再帰的に呼び出すことで評価し，結果 py_val_t 値を得る (f とする).
2. 引数の位置にある式 $E_1, \cdots, E_n$ をすべて，評価器を再帰的に呼び出すことで評価し，py_val_t 値のベクタ (py_val_vec_t) を得る (A とする).

3. f の型が Python 関数 (`py_type_ifun`)，native 関数 (`py_type_nfun`) のどちらでもなければ実行時エラー．

4. native 関数の場合:

   (a) f が受け取る引数の数 (f`->u.n.arity`) と，今渡された引数の数 `py_val_vec_size(`A`)` の一致を確かめ，食い違っていれば実行時エラー．

   (b) 一致していれば，まずスタックトレースにエントリを追加する (`stack_trace_push`)．

   (c) 続いて，対応する C の関数 (f`->u.n.f`) に，引数を与えて呼び出す．関数内部で実行時エラーが起きたときのために，引数以外に，スタックトレースや現在のソース位置を渡す．また，関数内部で他の関数を呼ぶときのために，大域環境も渡す．まとめると，実際の呼び出しは，

   $$f\texttt{->u.n.f}(genv,\ bt,\ pos, A);$$

   のようになるだろう．ここに，genv, bt, pos はそれぞれ，大域環境，スタックトレース，その関数呼び出しのソース位置である．また，A を渡す際に，ベクタの形で渡す代わりに，0 番目の引数，1 番目の引数, $\cdots$ を取り出して関数に渡してやることもできる．たとえば引数が 3 つであれば，

   $$f\texttt{->u.n.f}(genv,\ bt,\ pos, A \text{ の第 } 0 \text{ 要素}, A \text{ の第 } 1 \text{ 要素}, A \text{ の第 } 2 \text{ 要素});$$

   のように．このようにしたほうが，C 関数内部での記述が簡単になる．一方このやり方は，引数の数に応じて場合わけをしなくてはならず，もちろんその場合分けの数もある一定数しかソースコード内にはかけないから，こうするのであれば，

   - 引数の個数がある一定個数 (例: 4 個) までの関数は，後者のやりかたで実際に 1 つの引数を C の 1 引数として渡す．
   - それより多い引数を持つ関数は，引数全体をベクタとして渡す．

   という規則を決めて場合わけを行うと良い．

   (d) 呼び出しからの復帰後，スタックトレースから頂上のエントリを削除する (`stack_trace_pop`)．

   (e) 呼び出しが返した値を返す．

5. Python 関数の場合:

   (a) 関数のパラメータベクタ (f`->u.i.ps`; P とする) の長さ (`str_vec_size(`P`)`) と，与えられた引数の個数 (`py_val_vec_size(`A`)`) の一致を検査する．一致していなければ実行時エラー．

   (b) 一致していれば，新しい局所環境を作り，関数各パラメータ P_i と，対応する引数を評価した値 A_i の間の対応を登録する．

   (c) スタックトレースにエントリを追加する (`stack_trace_push`)．

   (d) 作られた新しい環境を局所環境として，関数本体 (`stmt_t` のベクタ; f-¿u.i.b) を評価する．これは，`eval_stmt_vec` という関数を別途作り，それを呼び出す．

(e) スタックトレースから頂上のエントリを削除する (stack_trace_pop).

(f) eval_stmt_vec が返してきた値 r に応じて，適切な値を返す．詳しくは下を見よ．

eval_stmt_vec という下請け関数は，基本は文の並び (stmt_t のベクタ; stmt_vec_t) を順に評価するものである．通常は最後まで文を順に実行していくことになるが，途中で return 文が実行された場合は直ちに実行を打ち切らなくてはならない．他にも，continue, break についても同様の処理をしなくてはならない．具体的には，eval_stmt_vec(S，$lenv$，$genv$，bt) は，S 中の文を順に評価していくが，それに加えて以下の動作をするものとする．

- 途中で return E が実行されたら，E を評価した値を直ちに返す.
- 途中で continue が実行されたら，それを示す特別な py_val_t 値 py_val_continue を返す.
- 途中で break が実行されたら，それを示す特別な py_val_t 値 py_val_break を返す.
- そのどれも実行されずに最後の文まで実行を終了したら，それを示す特別な py_val_t 値 py_val_next を返す.

これによると，上記の最後の部分:

> r に応じて，適切な値を返す

とは以下のことである.

- r が，通常のデータをあらわす py_val_t 値であれば，r をそのまま返す.
- r が py_val_continue または py_val_break であればエラー (ループ外で continue/break 文の実行は違法)
- r が py_val_next であれば，py_val_none を返す．これは，return 文を実行しなかった関数は None を返すという Python の仕様にしたがっている.

E_0 が属性参照 ($E.f$) 式の場合の例外事項: 以上で関数呼び出しの評価方法がわかった．しかしひとつだけ，mini-Python において導入した特別規則を取り扱うために，修正が必要である．それは，関数の位置 (E_0) に，属性参照 ($E.f$ の形の式) が来た場合である．たとえば,

```
L.append(1)
```

のような式で，これは関数呼び出し式であって，E_0 =L.append, E_1 =1 というものである.

この式を上で述べた規則に従って評価すると，まず L.append を評価しにいくことになるが，我々は一般の属性参照式をサポートしているわけではないので，これは評価できない．その代わりに，上の式は形式的に,

```
append(L, 1)
```

と同じであるとみなす．一般に，

$$E_0.f(E_1, \cdots, E_n)$$

を

$$f(E_0, E_1, \cdots, E_n)$$

と同じであるとみなすわけである．そのために，上で述べた E_0 を評価する部分の規則を以下のように拡張しておく必要がある．

1. E_0 の種類が `expr_kind_attref` 以外の場合は前述したとおり，E_0 を評価した結果を f とする．
2. E_0 の種類が `expr_kind_attref` である場合は，そこで参照されているフィールド名 E_0`->u.atr.f` を変数名として環境を探索し，それを f とする．同時に，引数として渡される式の並びを，$E_0, \cdots, E_n$ として，以降のステップへ進む．

それ以降は，上で述べた規則の 2. 以降を用いればよい．

4.7 演算子

4.7.1 各種演算子の意味

mini-Python に登場する演算子は以下のものがある．

論理積・和: `and`, `or`,

論理否定: `not`,

要素判定: `in`, (`not in`),

比較 (identity): `is`, (`is not`),

比較 (値): `==`, `!=`,

大小比較: `>`, `>=`, `<`, `<=`,

和・差．単項プラス/マイナス: `+` (infix と prefix), `-` (infix と prefix),

積・商・剰余: `*`, `/`, `%`,

ビット反転: `~`,

ビットシフト: `<<`, `>>`,

ビット積・和・排他和 `&`, `|`, `^`

ほとんどの演算子の意味は記号と名前から想像できるとおりだろうが，いくつか注意が必要な演算子もある．それらについては，付録 B にまとめておいた．

4.7.2 論理積，和以外の演算子の評価方法

論理積，和の評価の仕方は少し例外的なので後述するが，それ以外のものはどれも，それぞれの演算を実現する関数を用意しておいて，それへの関数呼び出しとみなして実行すればよい．具体的には以下のようである．

- 演算子の種類 (token の種類をあらわすシンボルで表現すればよい) と，それが単項演算子であるか二項演算子であるか (`fix_kind_t` 型の値で，`fix_kind_prefix` または `fix_kind_infix` のどちらか) を受け取り，それに対応する関数名を返す関数をつくる．

  `char * operator_fun_name(token_kind_t` k`, fix_kind_t` fix`)`

  それぞれの関数を，次のいずれかの方法で実現する．

  - 処理系内部のどこかに C の関数として定義し，それを native 関数として，処理系立ち上げ時に作り (`mk_py_nfun`)，大域環境に登録する．
  - 組み込みライブラリを記述した Python プログラムを作り，その中に定義を記述する．

  ひとつの演算子の動作も実は案外複雑である．たとえば上述したとおり，+演算子ひとつでも，数同士の場合，文字列同士の場合，$\cdots$ と様々な場合がある．これを全部 C の関数の中で実装するのは非常に苦労する．全体として楽をするために，なるべく多くの部分を Python で記述し，その中で適宜，どうしても必要なところだけ小さな native 関数を作って呼び出すようにするとよい．付録 C に，例をあげておいた．

どちらで実装されているにせよ，この準備が整ったら演算子式の評価自体は簡単である．

1. `operator_fun_name` によって，対応する関数名を得る．
2. 関数名を環境中で検索し，関数をあらわす `py_val_t` 値を得る (f とする)．
3. 後は，前節の「関数呼び出し」の評価方法の 2 以降と同じ処理をする．

したがってコードを書く際は,「関数呼び出し」の評価方法の 2 以降の部分をひとつの関数として作り，両者から呼び出すように書くのが良い．

参考までに，図 1 には演算子と関数名の対応例を書いておく．もちろんこれを律儀にまねする必要などまったくないが...

`ntv_is` という関数の先頭にある `ntv_`は，処理系内部の慣習として，それが native 関数として実装されていることを示す印である．つまり，それ以外の演算子はすべて Python 関数として定義されている．

4.7.3 論理積，和の評価方法

論理積 (`and`) の意味は a `and` b は,

- a が「偽」をあらわすデータであれば a を返す．

演算子	関数名	演算子	関数名
not	not_	- (infix)	sub
is	ntv_is_	- (prefix)	prefix_sub
in	contains	*	mul
==	eq	/	div
!=	ne	%	mod
>	gt	~	invert
>=	ge	<<	lshift
<	lt	>>	rshift
<=	le	^	xor
+ (infix)	add	&	and_
+ (prefix)	prefix_add	|	or_

表 1: 演算子と関数名の対応

- さもなければ b を返す

というものである．論理和 (or) の意味は a and b は，

- a が「偽」をあらわすデータでなければ a を返す．
- さもなければ b を返す

ここで mini-Python においては (Python を踏襲して)，0, 空文字列 (""), None の 3 つを「偽」をあらわすデータと定めている．

さて，これだけではなぜ and, or を他の演算子と同様に実装できないのかは不明である．それは，and, or は「短絡的評価」といって，a を返す場合，そもそも b を評価すらしない，という規則があるからである．つまりたとえば，

```
def f(x):
    print "f"
```

のような定義をしておいて，

```
0 and f(3)
1 or f(5)
```

などの式を実行しても，"f"は表示されない．これは,「まずすべての引数を評価し，それらの値を元に式全体の値が決まる」という，その他の演算子の評価方法とは違うことに注意．

そこで，and, or だけは特別に，短絡的評価を行うコードを別途書くことにする．and であれば，

1. 第 1 引数を，評価器を再帰的に呼び出して評価し，py_val_t 値を得る (a とする)．
2. a が偽であれば直ちにその値を返す．
3. さもなければ第 2 引数を，評価器を再帰的に呼び出して評価し，その値を返す．

もはや or を説明する必要はないであろう．

4.8 添え字 (∗)

添え字式 ($E_0[E_1]$ は，ほとんど演算子と同じで，たとえば `getitem`(E_0, E_1) という関数呼び出しとみなせばよい．評価方法は，

1. 関数名 (たとえば `getitem`) を環境中で検索し，関数をあらわす `py_val_t` 値を得る (f とする)．
2. 後は，前節の「関数呼び出し」の評価方法の 2 以降と同じ処理をする．

4.9 属性参照

先にも述べたとおり，この式自体を単独で評価することはありえない．単に実行時エラーを起こせばよい．関数呼び出しの関数の位置にこれが現れる場合については，すでに述べた関数呼び出し式の評価において特別に処理されている．

属性参照式が，不正な位置に現れないかどうかの検査は，構文解析中に行うことも可能であるが少し面倒である．この説明では，実行時に検査することとしている．

5 文の評価

5.1 概要

文には以下の種類があった．

- 式文 (`stmt_kind_expression`)
- 代入文 (`stmt_kind_assignment`)
- del 文 (`stmt_kind_del`)
- pass 文 (`stmt_kind_pass`)
- return 文 (`stmt_kind_return`)
- break 文 (`stmt_kind_break`)
- continue 文 (`stmt_kind_continue`)
- print 文 (`stmt_kind_print`)
- global 文 (`stmt_kind_global`)
- if 文 (`stmt_kind_if`)
- while 文 (`stmt_kind_while`)
- for 文 (`stmt_kind_for`)

- fundef 文 (stmt_kind_fundef)

それに応じて我々の作る eval_stmt の概形はひとまず以下のようになるだろう.

```
py_val_t eval_stmt(stmt_t stmt, env_t lenv, env_t genv, stack_trace_t bt)
{
  switch (stmt->kind) {
  case stmt_kind_expression:
    ...
  case stmt_kind_assignment:
    ...
  case stmt_kind_pass:
    ...
  case stmt_kind_return:
    ...
  case stmt_kind_break:
    ...
  case stmt_kind_continue:
    ...
  case stmt_kind_del:
    ...
  case stmt_kind_print:
    ...
  case stmt_kind_global:
    ...
  case stmt_kind_if:
    ...
  case stmt_kind_while:
    ...
  case stmt_kind_for:
    ...
  case stmt_kind_fundef:
    ...
  default:
    ...
  }
}
```

eval_stmt の返り値の型は py_val_t で，これは以下の値を返す約束である.

- その文が return 文 (return E) を実行して評価を終了したならば，E を評価した py_val_t 値．E が省略された場合は return None とみなされる.

- その文が continue, break 文を実行して評価を終了したならば，それぞれ py_val_continue, py_val_break.
- その文がそれ以外によって評価を終了したならば，py_val_next.

それぞれの文の評価方法は思ったよりはきっと簡単である．最後の 4 つ以外は当たり前といっても良いようなものである．

5.2 式文

単にその式を評価し，py_val_next を返せばよい.

5.3 代入文

代入文の文法は，x = E (x は変数) または $E_0[E_1]$ = E の形をしている．前者は変数への代入であり，後者は E_0 をひょうかした結果のリストの要素変更または辞書の要素の変更・追加である．

変数への代入 x= E の場合: 1. E を評価し，結果を v とする．

2. 現在の局所環境で x を探索 (env_lookup) し，x が大域変数である (py_val_global が返された) ならば，大域環境中の x を v にセット (env_set) する．さもなければ，局所環境中の x を v にセットする．そして py_val_next を返す.

   2 節で述べた仕組みによって，この式がトップレベル (関数定義の外) で実行された場合，上で「局所環境」と書いたところが実は大域環境のことである場合もあることに注意.

リスト・辞書への代入 $E_0[E_1]$= E の場合 ($*$): この代入文を，関数呼び出し setitem(E_0, E_1, E) であるとみなして評価する．つまり，4.8 節の，添え字式の評価と同じようなことを行えばよい．そして py_val_next を返す.

5.4 del 文 ($*$)

del $E_0[E_1]$ を評価するには，この del 文を関数呼び出し delitem(E_0, E_1) とみなして評価する．そして py_val_next を返す.

5.5 pass 文

py_val_next を直ちに返せばよい.

5.6 return 文

return E (E が省略されていたら，None とみなす) を評価するには，

1. E を評価する
2. その結果を返す.

5.7 break 文

py_val_break を返す.

5.8 continue 文

py_val_continue を返す.

5.9 print 文

print E は，print_(E) という関数呼び出しとみなして評価する.

5.10 global 文

global x の評価は，現在の局所環境において，x の値を py_val_global にセットする．ただし，すでに x の値が py_val_global 以外の値にセットされていたら，それはエラーである．つまり，すでに局所変数である x を後から大域変数にすることはできない.

5.11 if 文

if 文の文法は，if E_0: S_0 elif E_1: $\cdots$ というものだが，要するに,「条件 $\Rightarrow$ 実行すべき文の列」というものがいくつも並んでいるものである．したがって評価方法は,

条件部を評価し,「偽」と評価されなければ対応する文の列を評価したものを返す

と言うだけに過ぎない．偽と評価されたら次の条件部を評価する．どの条件部も偽と評価されたら，py_val_next を返せばよい.

「偽」と評価されるというのは，and, or のところと同じで，0, None, ""のいずれかに評価されるということである.

文の列の評価 if 文，while 文，for 文，def 文において，文の列 (stmt_t のベクタで，stmt_vec_t) を評価する必要が生ずる．それを行うために,

py_val_t eval_stmt_vec(S, $lenv$, $genv$, bt)

という関数を作って，各所で再利用する．この関数の動作は単に,

1. 各文を順に評価する

2. ある文の返り値が py_val_next であれば次の文を評価する

3. py_val_next 以外であれば，直ちにその値を全体の返り値として返す

4. 最後まで py_val_next 以外の値を返す文がなかったら，py_val_next を全体の返り値として返す

5.12 while 文

while 文の文法は while E: S (E は式，S は文の列) で，その評価方法は，

1. E を評価する．偽であれば直ちに py_val_next を返す

2. 偽でなければ S を評価する．その返り値に応じて以下の動作をする．

   - py_val_next：S が最後まで continue/break/return を実行することなく評価を終了した．したがってもう一度 1 に戻る．
   - py_val_continue：S が continue を実行して評価を終了した．したがってもう一度 1 に戻る．
   - py_val_break：S が break を実行して評価を終了した．評価を終了して py_val_next を返す
   - Python データに対応した値：S が return を実行して評価を終了した．評価を終了してその値を返す．

5.13 for 文 (∗)

for 文の文法は for x in E: S (E は式，S は文の列) である．E は文字列，タプル，リストのどれかに評価されなくてはならない．

for 文も while 文と同じループなので，やり方はほとんど同じである．各自考えること．

5.14 def 文

def 文は関数を定義する文で，実際のプログラムではほとんどがトップレベルに書かれるが，少なくとも文法上は，文を書ける任意の場所に書くことができる．形式上は def 文も，その場で (他の文と同様)「実行」されているのである．

def 文を実行したときに起こすべき効果は，代入文のそれに似ている．つまり，環境中に，新しい，「変数名 → 値」の組を登録する．値の方は，つまりそこで定義された Python 関数である．pyvalues.h に定義されているとおり，Python 関数は py_ifun という構造体であらわされていて，定義されたときの名前，引数のリスト (文字列のベクタ)，本体 (文の列) を格納している．def 文からそれらの内容を取り出して環境を書き換えればよい．

6　プログラム (file_input) の評価

プログラム (1 つのファイル全体を構文解析した結果) は，つまりは単なる文の列なので，`eval_stmt_t` を使って評価すればよい．ただし，返り値が `py_value_next` 以外であれば実行時エラーである．具体的には，

- 返り値が `py_value_continue`, `py_value_break` であれば，「ループ外で continue/break が実行された」というエラー
- Python データに対応する値であれば，「ループ外で return が実行された」というエラー．

7　インタプリタの main 関数

ある mini-Python プログラムのファイルを実行すべくインタプリタが起動された際，基本的には，そのファイルを，1 節で述べた `exec_file` を用いて実行すればよいのだが，これまでに述べたことを総合して，その前にいくつかの準備が必要である．以下にまとめておく．

1. 空の大域環境を作り，そこに，native 関数を登録する．
2. 空のスタックトレースを作る
3. 組み込み関数が定義されている Python ファイルを，ユーザが与えたファイルに先立って (`exec_file` を用いて) 実行する．

A　代表的実行時エラー

未定義変数の参照 (存在しない変数の参照): たとえば，

```
def f(x):
  return y + 1
```

における `y` や，

```
def f(x):
  return g(x)
```

における `g` など．

型エラー (型の正しくない操作): 関数でないデータを呼び出そうとする，文字列，タプル，リスト，辞書のいずれでもないデータに対して添え字参照 (`a[x]` のような式) を行う，+演算のオペランドが不正である，などの意味のない操作である．例えば，

```
def g(f, x):
  return f(x)

g(1, 2)
```

における `f(x)` は，`f` が関数でない値 (1) であるために不正であるし，

```
def concat(a, b):
  return a + b

concat("hello", [1, 2, 3])
```

における `a + b` は，文字列とリストを足し算演算子で結んでいるために，不正である．

不正な添え字: 文字列，タプル，リスト，辞書中のデータを，添え字で参照する際に，添え字が不正である．つまり，文字列，タプル，リストに対して整数以外の添え字が渡された，長さ以上の整数が渡された，辞書に対して，辞書中のキーとして存在しない値が添え字として渡された，などの場合である．例えば，

```
a = [ 1, 2, 3]
a[4]
```

や，

```
a = { 0 : 1, 2 : 3 }
a[1]
```

などがそれに該当する．

注: Python 辞書　Python における「辞書」というデータ構造は非常に強力である．添え字として整数だけでなく，文字列などを含めた様々なデータを許すので，一般的なマッピング (キーとそれに付随する値の組) を手軽に実装することができる．

しかし，添え字として任意の Python データを許すわけではない．許されるのは一言で言えば，変更不可能なデータである．mini-Python においては整数，浮動小数点数，None, 文字列，タプル，関数は変更不可能であり，リストと辞書は変更可能である．タプルの要素としてリスト，辞書が含まれている場合 (例: (1, 2, [3, 4])) も，これは変更可能であるとみなす．すなわち，mini-Python データのうち，Python 辞書のキーとして許されるのは，以下のものである．

- 整数，浮動小数点数，None, 文字列，関数は辞書のキーとして許される．
- 辞書のキーとして許されるデータのタプルもまた，辞書のキーとして許される．

このような仕様になっているのには理由があるのだが，皆さんはあまり気にせずに，**すくなくとも上記のデータをキーとして許せば**，それ以外のデータ (リスト，辞書) をキーとして許すかどうかは任意とする．

B　注意が必要な演算子

B.1　identity 比較

is という演算子は，a is b によって，a と b が「同じ」かどうかを判定する (同じなら 1, 違うなら 0 に評価される) が,「同じ」という意味に若干の注意が必要である．これは，プログラム言語の世界ではしばしば identity の比較と呼ばれているもので，値として一見同じものでも，identity は異なるということがある．mini-Python においては以下のように定義する．

- None 同士の比較は 1
- 整数，浮動小数点数，同士の比較においては，それらが値として等しい場合に 1
- 文字列同士の比較は，文字の列として等し (長さが同じで，同じ場所にある文字が同じ) ければ 1
- タプル，リスト，辞書，関数同士の比較においては，それらが同じ場所で作られたデータであれば 1
- それ以外の場合 0

主に注意が必要なのは 4 番目の要素である．つまり，同じ (1, 2) というタプルでも，別の場所で生成されたものであればそれらは異なるわけである．最も単純な例が,

```
>>> (1, 2) is (1, 2)
0
```

ということである．我々のデータ表現においては，タプル，リスト，辞書，関数同士の is による比較は py_val_t 値がポインタ (アドレス) として等しいかどうかを調べればよい．逆に，py_val_t 値がポインタ (アドレス) として異なっていても，なお is によって 1 が返される可能性があるのが，文字列同士，浮動小数点数同士の比較の場合である．それらはポインタとしてことなる二つの値でも，is によって 1 を返す場合が存在する．

B.2　値の比較 ($*$)

注:　値として整数，浮動小数点数，None, 関数しかなければ，==と, is は同じ意味になる．

==という演算子は，基本的には，a == b によって a と b が「値として」等しいかどうかを判定する．タプル，リスト，辞書同士が値として等しいとは，その中に含まれている要素が，やはり値として等しいということである．つまり,

```
[1, 2] == [1, 2]
{1: 2} == { 1:2}
[1, [2,3]] == [1,[2,3]]
```

などは皆，1 を返す.

ただし，異なる場所で定義された関数同士が「(写像として) 同じ関数であるかどうか」を返すかどうかは判定不能なので，関数同士の比較においては，is と同様，同じ場所で定義されたものである場合だけ，等しいとみなす.

B.3 大小比較

大小比較の演算子 (>, >=, <, <=) は，数だけではなくあらゆる Python データ型に対して定義されている．数に対しては自然な比較を行う．文字列同士，タプル同士，リスト同士に対しては，辞書式に比較が行われる (∗)．それ以外の場合については，結果は「一貫していれば良い」とされている．つまり，同じもの同士を比較したときに常に同じ結果が出れば良い，とされている．詳しくは Python のリファレンスマニュアルの 5.9 節を参照.

B.4 単項+/-

和 (+)・差 (-) に関しては，同じ記号でも単項演算子として使われている場合と，2 項演算子として使われている場合があるので注意が必要である．どちらであるかの情報は構文木に書かれている.

B.5 列の連結するための+ (∗)

和に関してはさらに注意が必要で，必ずしも数同士の足し算に限らない．文字列，タプル，リストの連結のためにも使われる.

B.6 列の繰り返しのための* (∗)

*は数同士の掛け算とは限らず，(文字列，タプル，リスト) * 整数によって，それぞれの列を右辺で指定された回数だけ繰り返したものになる．つまり，

```
"tau" * 5
```

は，

```
"tautautautautau"
```

に評価される.

B.7 文字列置換のための%(∗)

剰余演算子 (%) は，数同士の剰余に限らず，文字列に値を埋め込む演算子としても使われる．たとえば，

```
"hello %s san" % "tau"
```

とすると，”体として，

```
"hello tau san"
```

という文字列に評価される．二つ以上の”プルとする．

```
"hello %s %s" % ("tau", "san")
```

も，

```
"hello tau san"
```

となる．右辺に与えるのは文字列 (またはそのタプル) とは限らない．任意の Python 値を与えてよい．たとえば，

```
a = 10
b = 20
"%s * %s = %s" % (a, b, a * b)
```

において最後の行の式は，

```
"10 * 20 = 200"
```

という文字列に評価される．

Python は，`%s` だけでなく，`%d`, `%f` やそれらに対する桁数指定のオプションなど複雑な表示のための指示をサポートしているが，mini-Python においては，`%s` だけを実装すればよい．

C　演算子の定義方法の実例

さて，数々の演算子式ならびに添え字式が結局ある関数の呼び出しとみなされると述べた．それらの関数は native 関数として処理系内部で直接定義されるか，Python 関数として定義され，その中で一部の処理が，native 関数呼び出しを使って実行される．どの部分を native 関数にするかの基準は，楽をするためにまずは，

> できるだけ Python で，Python で書けない所は native で

と思っておけばよい．ここでは，いくつかの実例を通して演算子の実現の方法を説明する．最初に簡単な動作をする演算子として，割り算 (`/`) を説明しよう．この演算子は両方のオペランドが整数である場合，整数での割り算 (整数の割り算は小数点以下切り捨て) を実行した値を返す．または両方のオペランドが整数または浮動小数点数で，少なくともどちらか一方が浮動小数点数である場合には，浮動小数点数の割り算を実行した値を返す．

今，a `/` b を，表 1 にあるとおり，`div`(a, b) という関数呼び出しとみなして実行しているとすると，その中身 (Python で定義した場合) は以下のようになるだろう．

```
def div(a, b):
    if ntv_is_int(a):
        if ntv_is_int(b):
            return ntv_div_int(a, b)
        elif ntv_is_float(b):
            return ntv_div_float(ntv_itof(a), b)
        else:
            type_error()
    elif ntv_is_float(a):
        if ntv_is_int(b):
            return ntv_div_float(a, ntv_itof(b))
        elif ntv_is_float(b):
            return ntv_div_float(a, b)
        else:
            type_error()
    else:
        type_error()
```

ここで，ntv_is_int, ntv_is_float は，それぞれある値が整数および浮動小数点数かどうかを検査する関数である．ntv_is_int は二つの引数がともに整数である場合に整数の割り算を行った値を返し，それ以外の場合は実行時エラーを発生する関数である．ntv_is_float はその浮動小数点数版である．ntv_itof は，引数が整数であればそれを浮動小数点数に変換して返し，それ以外の場合は実行時エラーを発生する関数である．ここに出てきた ntv_で始まる関数はいずれも，native 関数として定義されている．実際それらの値は処理系内部の C 関数として簡単に実装できるし，そうするのが自然である．

もちろん div 自体を native 関数で定義することも可能である．しかしおそらく，C で実装する部分は極力単純な関数だけにしておくほうが全体として疲れない．後に速度を向上させるために，よく使われる関数を native にする，ということは是非行うと良いが，まずはなるべく少ない手間で動かすのが先決と言うものだろう．

もっと複雑な例として，+演算子の場合を説明する．add は様々な引数の組み合わせを許す．

- 数 (整数または浮動小数点数) + 数 (整数または浮動小数点数)
- 文字列 + 文字列 (∗)
- タプル + タプル (∗)
- リスト + リスト (∗)

さらに数の場合は，割り算と同様の場合わけが必要になる．add 関数はたとえば以下のように書ける．

```
def add(a, b):
    if ntv_is_int(a):
        if ntv_is_int(b):
```

```
            return ntv_add_int(a, b)
        elif ntv_is_float(b):
            return ntv_add_float(ntv_itof(a), b)
        else:
            type_error()
    elif ntv_is_float(a):
        if ntv_is_int(b):
            return ntv_add_float(a, ntv_itof(b))
        elif ntv_is_float(b):
            return ntv_add_float(a, b)
        else:
            type_error()
    elif ntv_is_string(a) and ntv_is_string(b):
        return ntv_add_string(a, b)
    elif ntv_is_list(a) and ntv_is_list(b):
        ab = []
        for x in a:
            ab.append(x)
        for x in b:
            ab.append(x)
        return ab
    elif ntv_is_tuple(a) and ntv_is_tuple(b):
        return ntv_add_tuple(a, b)
    else:
        type_error()
```

string と tuple については，それぞれ ntv_add_string, ntv_add_tuple という native 関数で処理している．リストについては，append という，リストの後ろに一要素を加える関数 append があるので，それと用いて二つのリストをつなげたリストを作っている．その他に，ntv_is_tuple, ntv_is_list, ntv_is_string という，データ型を判定する native 関数も用いている．

この調子で，各種演算子を実装していく．

D Native 関数として登録しておくとよいであろう関数群

上記で述べたような各種演算子を実装するに当たって，いくつかの関数が native 関数として実装されることになる．それらは，演算子を実装しながら適宜付け加えて言っても良いのだが，見通しをよくするためにも，native 関数にふさわしいものを最初から用意しておくのが良いだろう．基本的には,「単純であり，かつそれらさえあれば他の演算子を実装できるはずの関数群」を native 関数として用意しておく．それらは大雑把に言って，特定の型への演算を実装する小さな関数群である．数は多いが一つ一つの関数の内部は単純である．

D.1　値が特定の型であるかを判定する関数群

- `ntv_is_int`(a)
- `ntv_is_float`(a)
- `ntv_is_string`(a) $(*)$
- `ntv_is_dict`(a) $(*)$
- `ntv_is_list`(a) $(*)$
- `ntv_is_tuple`(a) $(*)$
- `ntv_is_ifun`(a)
- `ntv_is_nfun`(a)

D.2　整数同士の演算に限定した各種演算子

- `ntv_add_int`(a, b)
- `ntv_sub_int`(a, b)
- `ntv_mul_int`(a, b)
- `ntv_div_int`(a, b)
- `ntv_mod_int`(a, b)
- `ntv_cmp_int`(a, b)
- `ntv_eq_int`(a, b)
- `ntv_ne_int`(a, b)
- `ntv_gt_int`(a, b)
- `ntv_ge_int`(a, b)
- `ntv_lt_int`(a, b)
- `ntv_le_int`(a, b)
- `ntv_invert_int`(a)
- `ntv_and_int`(a, b)
- `ntv_or_int`(a, b)

- ntv_xor_int(a, b)
- ntv_lshift_int(a, b)
- ntv_rshift_int(a, b)

例としてひとつだけ，ntv_add_int の定義を示す．

```
py_val_t ntv_add_int(env_t genv, stack_trace_t bt, src_pos_t pos,
                     py_val_t a, py_val_t b)
{
  return mk_py_int(py_val_int(a, bt, pos) + py_val_int(b, bt, pos));
}
```

注意点としては，

- すべての native 関数は本来の引数のほかに余分な引数として大域環境，スタックトレース，ソース位置を受け取る．4.6 節で述べた native 関数呼び出しの方法に対応している．
- 引数も返り値もすべて py_val_t 型で，我々が定めたデータ表現にのっとった形式で引数を受け取り，データ表現にのっとった形式で返り値を返す．
- 上記において，型の検査を明示的に行っているようには見えないが，それらは py_val_int 内部で行われている．この関数は，py_val_t 型の値として渡された値を，それが表す C の整数に変換する関数で，その値が Python の整数でなければ型エラーを発生させる．したがって上記のように書いただけで，結果的に型検査をしていることになる．

その他の関数もほぼ同様に，1, 2 行程度で実装できる．

ntv_cmp_int(a, b) は，二つの Python 整数を受け取り，$a < b$, $a = b$, $a > b$ のいずれであるかに応じて，それぞれ Python 値-1, 0, 1 を返す関数である．

D.3 浮動小数点数同士の演算に限定した各種演算子

- ntv_add_float(a, b)
- ntv_sub_float(a, b)
- ntv_mul_float(a, b)
- ntv_div_float(a, b)
- ntv_cmp_float(a, b)

ntv_cmp_float は ntv_cmp_int(a, b) と同じもので，ただし，a, b が浮動小数点である．

D.4 文字列の演算子 (∗)

- `ntv_len_string`(s) (文字列 s の長さ) (∗)
- `ntv_getitem_string`(s, i) (文字列 s の i 番目の文字) (∗)
- `ntv_add_string`(s, t) (文字列 s と t の連結) (∗)

D.5 タプルの演算子 (∗)

- `ntv_len_tuple`(t) (タプル s の長さ) (∗)
- `ntv_getitem_tuple`(t, i) (タプル t の i 番目の要素) (∗)
- `ntv_add_tuple`(t, u) (タプル s と t の連結) (∗)

それぞれ文字列の，似た名前の関数と同様の働きを，タプルに対して行う．

D.6 リストの演算子 (∗)

- `ntv_len_list`(l) (リスト l の長さ) (∗)
- `ntv_getitem_list`(l, i) (リスト l の i 番目の要素長さ) (∗)
- `ntv_append_list`(l, x) (リスト l の末尾へ x を追加) (∗)
- `ntv_setitem_list`(l, i, x) (リスト l の i 番目の要素を x にする) (∗)
- `ntv_delitem_list`(l, i) (リスト l の i 番目の要素を削除) (∗)
- `ntv_pop_list`(l, i) (delitem と同じ．ただし，削除された値を返す) (∗)

D.7 辞書の演算子 (∗)

- `ntv_len_dict`(d) (辞書 d の要素数) (∗)
- `ntv_has_key_dict`(d, k) (辞書 d が k をキーとして持っていれば 1) (∗)
- `ntv_getitem_dict`(d, k) (辞書 d のキー k に対する値) (∗)
- `ntv_setitem_dict`(d, k, x) (辞書 d のキー k に対する値を x に) (∗)
- `ntv_delitem_dict`(d, k) (辞書 d のキー k に対する値を削除) (∗)
- `ntv_keys_dict`(d) (辞書 d のすべてのキーのリスト) (∗)
- `ntv_values_dict`(d) (辞書 d のすべての値のリスト) (∗)
- `ntv_items_dict`(d) (辞書 d のすべての (キー, 値) のリスト) (∗)

D.8　各種値を文字列に変換する関数群

これらは各種値を表示するために，文字列に変換する関数である．また，文字列に対する”

- `ntv__repr__int(`i`)` $(*)$
- `ntv__repr__float(`f`)` $(*)$
- `ntv__repr__ifun(`f`)` $(*)$
- `ntv__repr__nfun(`n`)` $(*)$

D.9　文字列を表示する関数

- `ntv_print_string(`s`)` $(*)$

注:　文字列を許さない処理系を作る場合は，これの代わりに，int, float を受け取りそれを表示する関数，空白を表示する関数，改行を行う関数，くらいが必要だろう．たとえば，

- `ntv_print_int(`i`)`
- `ntv_print_float(`f`)`
- `ntv_print_space()`
- `ntv_newline()`