B演習 (言語処理系演習) 資料 (3)
字句解析器

田浦

2006/10/30

1 概要

先週も説明したとおり，字句解析器の機能を抽象的に述べると，与えられた文字列からの「単語」(字句，トークン; token) の切り出しということになる．もちろん単語といっても，自然言語の単語に対応するわけではない．したがって以下では字句と言う言葉のみを使うことにする．

for, def のようなキーワードはひとつの字句である．(や，, なども，プログラムの構文を解析する上では，重要な，単独で切り出すべき文字列であるから，それらもひとつの字句とみなす．変数は n であろうが，n_failures であろうが，futsu_sonnna_nagai_hensuu_tsukawaneeyo であろうが，プログラムの構文を解析する上ではひとつの単語とみなすのが便利であるから，それらはまとめてひとつの字句とみなす．大きな字句の例としては，文字列リテラル ("hello" など) がある．厳密な定義は後で出てくるが，基本的にはダブルクオート二つで囲まれた文字列全体がひとつの字句である．

例を挙げると，

```
x = "this is a string literal"
```

のような文字列は，x, =, "this is a stringliteral" の 3 字句と見なす．

空白は字句とはみなさない．結果として，

```
x="this is a string literal"
```

も字句の列に直した時点で，上記の 3 つの字句の列とみなされる．これはこの後の構文解析を簡単にする効果がある．

Python において，コメントは行の先頭や途中に#を置くことで始まり，その行の終わりまで続く．コメントも字句とはみなさない．

今週の内容は，

- そもそも字句とは何かをしっかりと定義する
- ファイルから文字列を読み込み，それを字句に切り分けて，字句の列を生成していくことができるような部品「字句解析器」を作る

- 「字句解析器」を単独にテストする小プログラムをつくり，実際にいくつものPythonプログラムを字句に切り分けてみる．
- デバッガの使い方を覚える．
- デバッグの方法論 (精神論?) を覚える．

2 字句の定義

字句の定義を分類しながら述べよう．以下で

- typewriterfont (TOK_XXX)

のように書いてあるのは，そのtypewriterfontという文字列がまさにひとつの字句であることを示している．また，(TOK_XXX) は，その字句を，われわれの処理系の中でなんという名前 (実際には#defineまたはenumで定義される整数) で呼ぶかを書いたものである．たとえば，

- or (TOK_OR)

とあるのは，'o', 'r' という2文字からなる文字列がひとつの字句であり，それに対応して処理系内部でTOK_ORという整数を定義することを意味している．

2.1 演算子またはそれを構成する字句

- or (TOK_KW_OR)
- and (TOK_KW_AND)
- not (TOK_KW_NOT)
- is (TOK_KW_IS)
- in (TOK_KW_IN)
- = (TOK_EQ)
- == (TOK_EQ_EQ)
- != (TOK_NEQ)
- > (TOK_GT)
- >= (TOK_GEQ)
- < (TOK_LT)
- <= (TOK_LEQ)

- + (TOK_PLUS)
- - (TOK_MINUS)
- * (TOK_MUL)
- / (TOK_DIV)
- % (TOK_MOD)
- ~ (TOK_TILDE)
- << (TOK_LSHIFT)
- >> (TOK_RSHIFT)
- ^ (TOK_XOR)
- & (TOK_AMP)
- | (TOK_BAR)

2.2 None

- None (TOK_NONE)

2.3 文を構成する字句

- break (TOK_KW_BREAK)
- continue (TOK_KW_CONTINUE)
- pass (TOK_KW_PASS)
- return (TOK_KW_RETURN)
- del (TOK_KW_DEL)
- print (TOK_KW_PRINT)
- global (TOK_KW_GLOBAL)
- if (TOK_KW_IF)
- elif (TOK_KW_ELIF)
- else (TOK_KW_ELSE)

- `for` (TOK_KW_FOR)
- `while` (TOK_KW_WHILE)
- `def` (TOK_KW_DEF)

2.4 括弧やカンマなど，いろいろなところで現れる記号

- `(` (TOK_LPAREN)
- `)` (TOK_RPAREN)
- `{` (TOK_LBRACE)
- `}` (TOK_RBRACE)
- `[` (TOK_LBRACKET)
- `]` (TOK_RBRACKET)
- `.` (TOK_PERIOD)
- `,` (TOK_COMMA)
- `:` (TOK_COLON)

2.5 リテラル

さて，ここまでの字句はすべて，ひとつの文字列が完全にひとつの字句に対応していた．しかしたとえば，`123`, `245.0`, `"abc"` などの，プログラムの文面に直接表れる値 (プログラム言語の用語ではしばしば，リテラル; literal と呼ばれる.「(プログラムに現れた) 文字通り」と言う意味である) については，ひとつの文字列で記述することは当然ながらできない.

整数をあらわすリテラル (整数リテラル) には，`1`, `123`, `-234`, $\cdots$ などの，文字通り無限個の文字列が含まれる．整数リテラルである文字列はどのようなものであるか，正確にはどのように述べられるだろうか．まずは以下のような自然言語による記述が可能であろう.

- `0` のみからなる文字列，または
- `0` 以外の数字 (`1`, $\cdots$ `9`) から始まり，以降，任意個の数字 (`0`, $\cdots$ `9`) が続く文字列

この自然言語記述から，整数リテラルを認識するプログラムを書くことも可能かもしれないが，もう少し厳密な表記法と，それに基づいた，ある程度機械的なプログラムの導出ができるのが望ましい.

そのような文字列の集合を規定するひとつの方法に，**正規表現 (Regular Expression)** があるということを知っていることだろう．以下ではこの記法を用いてそれぞれも字句とみなされる文字列の集合を記述する.

- 整数リテラル (TOK_LITERAL_INT)

  0 | (1|⋯|9)(0|1|⋯|9)*

  一般に正規表現という表現方法では，$X*$ は X の 0 回以上の繰り返し，$X+$ は 1 回以上の繰り返し，$X?$ は 0 回または一回の出現を表す．$(X_1|\cdots|X_n)$ は，X_1 から X_n までのどれか，という意味である．XY のように並べて書くと，まず X が出現し，その後に Y が出現するという意味である．

  したがって上記全体で，意味としては，

  - 0 というただ一文字の文字列，または，
  - $1,\cdots,9$ のどれかがまず出現し，その後に $0,\cdots,9$ のどれかが 0 回以上出現したもの

  という意味になる．

  注意としては，これは負の整数リテラル-234 を含んでいない．これを含むように定義をするには，

  -? (0 | (1|⋯|9)(0|1|⋯|9)*)

  とすればよい．

  負の整数リテラルを含まないとすると，-234 は，234 に単項演算子-を適用している式とみなされる．前者を採用すると字句解析器は若干簡単になる．

  また，03 のような文字列は違反であることにも注意．オリジナルの Python ではこれは 8 進数とみなされる．また，オリジナルの Python では 0x1a のような 16 進数の表現も認められている．mini-Python では簡単のため，十進表記のみを必須とする．もちろん，余力があればこれらも受け付けるようにしてみよう．

- 浮動小数点数リテラル (TOK_LITERAL_FLOAT)

  (0|1|⋯|9)*[.(0|1|⋯|9)*]

  [⋯] は，⋯ がオプショナル (あってもなくても良い) であることを意味する．?記号を使えば，(⋯)?とも書ける．

  またそれは，

  ⋯ | ϵ

  とも書ける．ϵ は空文字列をあらわす．

  浮動小数点数リテラルの集合は，上の正規表現を見ても分かるように，整数リテラルの集合を含んでいる．それを含まないように上の正規表現を書き換えることも可能だが，今はこの正規表現には現れない規則として，どちらにも含まれるリテラルは整数リテラルに属すという規則を加えておく．

- 文字列リテラル (TOK_LITERAL_STRING)

"$(x \mid \backslash y)^*$"

ここで，x はバックスラッシュ(\)，2 重クオート (") 以外の任意の 1 文字，y は任意の 1 文字をあらわす．

たとえば，

```
"regular string"
```

は素直な文字列リテラルである．中に"を含んだ文字列を作成したければ，

```
"he said \"hello\" to me"
```

のようにする．バックスラッシュはエスケープ文字と呼ばれ，文字列を構成する文字とはみなされない．上記の文字列は，中身としては，

```
'h', 'e', ' ', 's', 'a', 'i', 'd', ' ', '"', 'h', 'e', 'l', 'l', 'o', '"',
' ', 't', 'o', ' ', 'm', 'e'
```

の 21 文字を含む文字列である．バックスラッシュ自身を文字列に入れたければ，\\を使う．

バックスラッシュ+1 文字の組み合わせは，基本的にはバックスラッシュの後ろの文字とみなされる．たとえば，\qは，qと同じことである．しかし，バックスラッシュ後の一文字が以下のどれかである場合，特別な解釈を行う．

- 'n': Linefeed とみなされる (C の'\n' で表現される文字)
- 'r': Carriage Return とみなされる (C の'\r')
- 't': タブとみなされる (C の'\t')

本物の Python にはそのほかにも様々な特別解釈をする文字があるが，それらはオプションとしよう．ここでは上で述べたエスケープ文字を含む文字列を正しく処理できることを必須課題とする．

2.6 識別子

識別子 (TOK_ID) は以下の正規表現で表される文字列の集合である．

$(a \mid \_)(a \mid d \mid \_)^*$

ここで a はアルファベット (大文字・小文字)，d は数字 (0|$\cdots$|9) をあらわす．つまりは先頭はアルファベットまたは_ (アンダースコア)，それ以降は，アルファベット，_, または数字，ということである．

識別子 (identifier) とは，変数名，関数名などの，プログラム中に現れる名前のことである．

2.7 入力終端記号

プログラムが終端に達したとき，それを示す字句 (TOK_EOF) が生成される．

2.8 改行

ほとんどの言語では，改行は空白と同じとみなされる．したがって改行自身が字句として扱われることはない．同様に字下げも単なるプログラマにとって見やすくする慣習に過ぎず，字句解析の段階で捨てられる．

しかし Python は改行と字下げを文法の一部とみなすので，字句解析も適切にそれらを字句として生成してやる必要がある．Python においては，文法上規定された場所では，改行が行われなくてはならず，それ以外の場所での改行は許されない．文法上規定された場所には以下などがある．

- for x in E : の :の後ろ
- if E: の :の後ろ
- def f ($x, y, \cdots$): の :の後ろ
- 単純な文の後ろ．単純な文とは，print のような，内側に他の文を含まない文のことである．つまりたとえば，print x の後ろでは改行が行われなくてはならない．

正確にはすべて来週以降，mini-Python の文法全体を定義するときに明らかになる．そしてそれ以外の場所，たとえば式の途中などでの改行は許されない．たとえば，

```
x = 1 +
    2
```

のような書き方は Python 以外のほとんどの言語では「変わった趣味」に過ぎないが，Python では文法エラーとなる．

字句解析器の役割は改行が現れた際にそれを，字句として生成することである．

似たこととして，入力が終わりまで達したら，それを示す特殊な字句を生成する．

改行には TOK_NEWLINE, 入力終端には TOK_EOF という名前の字句を生成することとする．

2.9 字下げ

通常のプログラミング言語では字下げは単なる好みの問題であるが，Python では，字下げが文法の一部とみなされるということから，字句解析器も字下げを擬似的な字句として生成することにする．たとえば for 文を含んだ文の一例は，

```
for x in S:
  s = s + x
  p = p * x
print s
print p
```

である．字句解析器はこれに対して以下のような字句の列を生成するようにする (以下の newline, indent, **dedent** に着目).

`for` | identifier (`x`) | `in` | identifier (`S`) | `:` | **newline**

indent | identifier (`s`) | `=` | identifier (`s`) | `+` | identifier (`x`) | **newline**

identifier (`p`) | `=` | identifier (`p`) | `*` | identifier (`x`) | **newline**

dedent | identifier (`print`) | identifier (`s`) | **newline**

identifier (`print`) | identifier (`p`) | **newline**

2行目が始まる際に，indentという字句が生成され，そこで「1段深い字下げ」が行われたことを示す．これは実際に何文字の空白が挿入されたかとは関係ない．一方，3行目の最初にはindentは生成されない．これは，上の行と字下げレベルが同じだからである．そして，4行目の最初で，dedentを生成し,「字下げが一段浅いレベルに戻った」ことを示す.

字句解析器が実際にどのような規則でindent/dedent字句を生成したらよいかについて，Pythonの Language Reference (邦訳: リファレンスマニュアル)2.1.8節には以下のように記述されている．われわれもこれに従う.

> . . . The indentation levels of consecutive lines are used to generate INDENT and DEDENT tokens, using a stack, as follows.
>
> Before the first line of the file is read, a single zero is pushed on the stack; this will never be popped off again. The numbers pushed on the stack will always be strictly increasing from bottom to top. At the beginning of each logical line, the line's indentation level is compared to the top of the stack. If it is equal, nothing happens. If it is larger, it is pushed on the stack, and one INDENT token is generated. If it is smaller, it must be one of the numbers occurring on the stack; all numbers on the stack that are larger are popped off, and for each number popped off a DEDENT token is generated. At the end of the file, a DEDENT token is generated for each number remaining on the stack that is larger than zero. . . .

つまり，ある行の字下げが，直前の行よりも深ければそこでindent字句がひとつ生成される．これをindentが「一段」深くなる，と呼ぶことにしよう (ここで何文字字下げが深くなったかは関係ない．直前行よりも深いということだけが重要である).

直前行と字下げが同じならば何も生成されない.

浅ければ，いくつかのdedent字句が生成される．いくつであるかは，現在の行の字下げ量が「何段前の」字下げ量と同じかによって決まる．1段前であればひとつ，2段前であれば二つ，のdedent字句が生成される.

このようにしてindent/dedent字句を生成しておくと，後の文法規則の記述は簡潔なものになる.

上の例を，indent/dedentだけを補うと，以下のような感じになる.

```
for x in S:
->   s = s + x
->   p = p * x
<- print s
print p
```

->が indent, <-が dedent 字句をあらわす.
もう少し複雑な例として,

```
for x in S:
  s = s + x
  if x:
    s = s * 2
  else:
    s = s * 3
print s
print p
```

に対しては,

```
for x in S:
->  s = s + x
    if x:
->      s = s * 2
<-  else:
->      s = s * 3
<-<- print s
print p
```

である. `print s` の前に二つの dedent が生成されていることに注意. これは直前の `s = s * 3` に対して, 2 段分字下げレベルが浅くなっていることを反映している.

2.10 コメント

mini-Python プログラム中にコメントを書くことができる. コメントは, 本質的でない機能と思うかもしれないが, 実際には非常に重要である. コメントを許さない処理系があったら生産性は恐ろしく落ちるだろう. 文字通りのコメント以外にも, プログラムを修正する際に各所をちょこっとコメントアウトしたりしながら修正することは誰もがよく行っていることだろう.
Python でも mini-Python でも, コメントは#記号ではじめ, その行の終わり (改行まで) 続く. たとえば,

```
for x in E:
    # do this for each of x
    do_this(x)    # ... but what if this fails?
```

のように．字句解析器がコメントをどう取り扱うべきかを示すと以下のようになる．

- 通常 (例: 上記 3 行目), その行のコメントから行の終わりの改行までに対して，ひとつの newline 字句を生成する．つまり，#から改行直前までの文字列が存在しなかったものとみなす．
- 例外として (例: 上記 2 行目)，ある行のコメント前に空白しかない場合，その行に対しては一切の字句を生成しない．つまり，改行を含めてその行全体が存在しなかったかものとみなす．

つまり全体として上のプログラム片は以下のようにみなされ，

```
for x in E:
    do_this(x)
```

以下のような字句列が生成される．

`for` | identifier (`x`) | `in` | identifier (`E`) | `:` | newline

indent | identifier (`do_this`) | `(` | identifier (`x`) | `)` | newline

3　字句解析器のインタフェース (外部インタフェース)

字句解析器のインタフェースそのものを述べる前に，それに付随するいくつかの定義を述べる必要がある．それらを順に述べていく．

3.1　字句の種類をあらわす型とたくさんの定数

前節で述べた字句の種類の字句を，C 言語内では，多数の TOK_XXX というシンボルを定義して区別することにしよう．最も簡単にはそれらに 0 から始まる通し番号をつけ，

```
#define TOK_KW_OR 0
#define TOK_KW_AND 1
...
```

のように C プログラム内で定義するというものである．しかしこれでは間違えやすいし，何かの都合で間に新しい字句を割り込ませたくなったらそれ以降の定義を 1 ずつずらすなどの作業が生ずる．こんなときに便利なのが，enum という型である．

```
enum {
  TOK_OR,
  TOK_AND,
  ...
};
```

と書くだけで，TOK_OR は 0 に，TOK_AND は 1 に，という具合に自動的に番号がつけられる．

```
enum {
  TOK_OR = 100,
  TOK_AND = 200,
  ...
};
```

のように自分で番号を決めることも可能である．

また，それらを代入できる変数の型は自然には int 型であるが，普通の整数と区別するために，新しく型の名前をつけておくとよい．具体的には，

```
typedef enum {
  TOK_OR,
  TOK_AND,
  ...
} token_kind_t;
```

のようにしておき，字句の種類をあらわす変数には，token_kind_t 型を使うと決めておく．

3.2 字句解析器

さていよいよ字句解析器のインタフェースを定義しよう．インタフェースの基本構造は先週作った char_stream と同じで，なにを単位として切り出して来るかが違うだけである．要素としては以下のように分類できる．

- ファイルを指定して字句解析器を作る関数
- 次の字句を読む関数
- 現在の字句に関する情報を返す関数
- そのほか，エラーメッセージを出力するための情報 (ファイル名，行番号，カラム位置など) を返す関数

たとえば基本は以下のようになるだろう．

- typedef struct tokenizer { ... } * tokenizer_t 字句解析器を表す構造体の定義．
- tokenizer_t mk_tokenizer(char * $filename$): 字句解析器を作る．$filename$ を開き，以降そこから字句の列を生成するような字句解析器を返す．ファイルが存在しなかったり読み出せなければ，0 を返すことにしておこう．
- token_kind_t tok_cur_token_kind(tokenizer_t t): 「現在の字句」の種類を返す.「現在の字句」とは，字句解析器が作られた直後は，ファイルの最初の字句のことであり，それ以降は，以下の tok_next が呼ばれるたびにファイル上の次の字句になる．

- `char * tok_cur_token_str_val(tokenizer_t` t`)`: 現在の字句の種類が文字列リテラルまたは identifier であったときに，その文字列 (文字列リテラルであれば内容となる文字列そのもの，identifier でればその名前) を返す.

- `int tok_cur_token_int_val(tokenizer_t` t`)`: 現在の字句の種類が整数リテラルであったときに，その値を返す.

- `double tok_cur_token_float_val(tokenizer_t` t`)`: 現在の字句の種類が浮動小数点数リテラルであったときに，その値を返す.

- `int tok_next(tokenizer_t` t`)`: 次の字句を読み込む．これによって,「現在の字句」が変化する.

以上が字句解析器の本質となる部分で，以下はエラーメッセージを適切に表示するための補助となる関数である.

- `char * tok_filename(tokenizer_t` t`)`: その字句解析器に結び付けられた (読み込み中の) ファイル名.

- `int tok_cur_line_no(tokenizer_t` t`)`: 現在の行番号.

- `int tok_cur_column_no(tokenizer_t` t`)`: 現在のカラム番号.

- `char * tok_cur_line_string(tokenizer_t` t`)`: 現在行の最初から，現在読んでいるカラムまでを文字列として返す．これはエラーメッセージを表示する際に，エラーが起きた行の内容を表示するために用意しておく.

- `char * tok_cur_token_string(tokenizer_t` t`)`: 現在の字句を文字列として返す．たとえば不正な字句がおきたときにそれを表示するために用いる.

- `char * tok_cur_token_kind_string(tokenizer_t` t`)`: 現在の字句の種類を文字列として返す．たとえば identifier (`TOK_ID`) であれば，`"TOK_ID"`という文字列を，`TOK_LITERAL_INT` であれば，`"TOK_LITERAL_INT"`という文字列を返す．純粋にテスト用である.

注: 今回も，これからもそうだが，ここで示すインタフェースに一字一句従う必要はまったくない．説明を具体的にするために，ひとつの reasonable なインタフェースを示しているに過ぎない．好みや，自分がベターだと思うものにどんどん変更・追加していってかまわない.

字句解析器は常に「現在の字句」(最後に読み込まれた字句) を保持しており，その字句に関する情報が `cur_token_{kind,str_val,int_val,float_val}` などの関数によって得られる.

`tok_next` は，現在の字句を捨てて，次の字句を読み出す．ファイルの終端に達したら `tok` は終端をあらわす字句になる.

次の字句が正しくない字句であった場合，エラーを表示し，プログラム全体を終了 (exit) するものとしておこう.

本質的な部分はこれだけで，あとはエラーメッセージを表示するための関数と，テスト用の関数である.

通常言語処理系は構文エラーが生じた際，どの字句の周辺でエラーが起きたか，また，エラーが起きたファイル名と行番号，その周辺の文字列を表示する．たとえば，

```
def fib(n):
    if n < 2:
        return n 2
    else:
        return fib(n - 1) + fib(n - 2)
```

のプログラムは 3 行目に構文エラーがある．一つ一つの字句は正しい字句だが，return, n, 2 という字句の並びは正しくない．構文解析器としては，2 という字句が来たところでエラーを検出することになるであろう．このような際に，エラーメッセージとして以下のようなものを表示するのが親切と言うものであろう．

```
fib.py:3: parse error near token ‘2’
fib.py:3:       return n 2
```

これを行うために，字句解析器としては日ごろから，現在のファイル名と行番号，現在の行の内容，現在の字句，などを保持しておく必要がある．

3.3 テストプログラム

字句解析器を単体でテストするために，ファイルを読み込んで，文字通りその中身を字句の列として表示するだけの簡単なプログラムを作ってみよう．それは以下のとおり簡単なものになるはずである．

```
int tokenizer_test(char * filename)
{
  tokenizer_t t = mk_tokenizer(filename);
  while (tok_cur_token_kind(t) != TOK_EOF) {
    print_cur_token(t);
    tok_next(t);
  }
  return 0;
}

int main(int argc, char ** argv)
{
  if (argc != 2) {
    fprintf(stderr, "usage : %s filename\n", argv[0]);
    exit(1);
  } else {
```

```
    tokenizer_test(argv[1]);
    exit(0);
  }
}
```

ただし print_cur_token(t) は現在の字句を，テストのために詳しく表示するもので，以下のようなものとしておこう．

- identifier とリテラル (整数，浮動小数点数，文字列) に対しては，その C におけるシンボル名と，その値の文字列を，以下のように表示する．

  C におけるシンボル名 (値の文字列表記)

  たとえば 5 とい整数リテラルに対しては，

  TOK_LITERAL_INT (5)

- それ以外の字句に対しては，その C におけるシンボル名のみを表示する．たとえば，プラス記号であれば，

  TOK_PLUS

  のように．

たとえば以下のように定義できる．

```
void print_cur_token(tokenizer_t t)
{
  char * C_name = tok_cur_token_kind_string(t);
  switch(tok_cur_token_kind(t)) {
  case TOK_ID: case TOK_LITERAL_INT:
  case TOK_LITERAL_FLOAT: case TOK_LITERAL_STRING:
    printf("%s (%s)\n", C_name, tok_cur_token_string(t));
    break;
  default:
    printf("%s\n", C_name);
    break;
  }
}
```

3.4 本当のテスト

演習用 HP に，上記のテストプログラムに与えるべきテストプログラム集と，その正解をファイルとしておいておく．上で決めたルールを厳密に守っていれば，皆さんが作成したプログラムの出力と，HP においてある正解はぴったり一致するはずである．以下のようにしてチェックをしてみてほしい．

```
prompt% ./minipy prob_X.py > my_ans_X
prompt% diff --ignore-all-space my_ans_X ans_X
prompt%
```

prob_X.py, ans_X.py は演習 HP からダウンロードしたファイルである．X は問題番号で置き換えられる．diff は Linux または cygwin 環境では必ずインストールされているコマンドで，ファイルの中身を比べるものである．このプログラムが一切出力をしなければ正解と言うことである．--ignore-all-space は，本来は不必要のものだが，Windows 環境と Linux 環境で改行コードが違ったりといった細かい違いを無視するためにつけておく．

今週の目標はともかくここで全問正解をするプログラムを作ることである．こうして部品を作ったら単体でテストをしておかないと，後々部品を積み重ねてプログラムが動かないときに，苦労が大きくなる．

4 字句解析器の中身の概要

インタフェースを述べた後で，実際にどう作るかについて，概要のみ述べておく．

4.1 アルゴリズム以前のソースコードの輪郭

字句解析の主要部分はほとんどが tok_next という関数の中にある．

まず tokenizer という構造体の中身をきっちりと定義して，現在の字句に関する情報を適切に蓄えるようにしておく．その前にひとつの字句をあらわすものを，やはり構造体をつかって定義しておくのがよいだろう．

```
/* 字句そのものの定義 */
typedef struct token
{
  token_kind_t k;
  union token_val
  {
    int       v_int; /* int/char/bool value */
    char *    v_str; /* string value */
    double    v_float; /* single float value */
  } v;
} token, * token_t;
```

```
/* 字句解析器の中身 */
typedef struct tokenizer
{
   ...          /* そのほかもろもろ */
  token tok;    /* 現在の字句を保持 */
} tokenizer, * tokenizer_t;
```

この前提で tok_next を非常に大雑把に書けば以下のようにかかれることになるだろう.

```
void tok_next(tokenizer_t t) {
  空白を読み飛ばす;
  switch (現在の文字) {
    case '(' : /* TOK_LPAREN に決定; */
      1 文字読み進める; t->tok.k = TOK_LPAREN; return;
    case ')' : /* TOK_RPAREN に決定; */
      1 文字読み進める; t->tok.k = TOK_LPAREN; return;
    ...
  }
}
```

t->tok.k = TOK_LPAREN という文の実行によって，字句解析器が外部に対して提供しているインタフェース「現在の字句が tok というフィールドに書かれている」が守られていることに注意.

基本的には tok_next という関数から戻る際にきちんと，tok に次の字句がセットされているように書けばよいのである (個々の関数がはっきりと定められた動作をするという部品化の原則を思い出す). そうすれば，現在の字句に関する情報を返すことは簡単であろう.

さて，上で書いた大雑把なコードはもちろん非常に大雑把でしかなく，書かれていない詳細がいくつもある.

1. 2 文字以上からなる字句 (!=など) もある.
2. 最初の 1 文字を見ただけではそれが単独の字句なのか，次の文字と組み合わさるのか分からない字句もある. たとえば=は次に再び= がくればあわせてひとつの字句 TOK_EQ_EQ になるし，そうでなければ TOK_EQ である. <なども同様である.
3. もちろん，リテラル (数，文字列) や identifier はもっと複雑である.
4. コメントも正しく読み飛ばす必要がある.
5. Python 固有の厄介さとして，indent/dedent 字句を正しく生成する必要がある.
6. エラーメッセージを表示するための関数群 (行番号やカラム番号の問い合わせ) も正しく動作させる必要がある. 特にそれらのためには，現在構築中の字句や，現在読み込み中の行などを正しく把握しながら文字を読み込まなくてはいけない. 特に，読み出した文字を読んだ端から捨てて行ってはいけない.

などである．

最後の項目を眺めていると，tokenizer を作るために，先週のウォームアップ課題で作った部品 char_stream を少し変更・拡張したものが使えそうだと気づくだろう．現在の行や現在の位置を把握するのは，char_stream という下請け部品にやらせるという方針にしよう．

このように,「一度に多くのことをやらない．なるべく独立した部品に分割する」という方針は,ひとつの部品を作る際にも適宜適用していくのである．最初は簡単だと思って作り始めた部品が,やってみると思ったより複雑ということは良くある．そういう時は，急がば回れでいったん当面の部品の実装を休んで,「その部品をより簡単に作るための下請け部品」を導入するのである．今回は char_stream 下請け部品に「現在位置 (ファイル名，行番号，列番号)，ならびに現在読み込み中の行」を把握するという仕事を丸投げすることで，tokenizer を簡単にするのである．ここで作る char_stream は先週のウォームアップ課題で作成したものに，若干の拡張をする必要がある．

identifier やリテラルなどの字句に関しては，字句の種類 (TOK_ID, TOK_LITERAL_INT など) だけではなく，その値 (identifier であればその文字列，数リテラルであればその整数としての値) も求める必要がある．どちらにしても下準備として「その字句を構成する文字列」を取り出さなくてはならない (それができればそれを数に変更するには atoi, atof などの C ライブラリを使えばよい)．つまり，次の字句の読み出しを始める前に空にして，一文字読み込まれるたびに追加をするような,バッファが必要である．

同じようなバッファは，char_stream 部品が「現在の行を構成する文字列」を把握するためにも必要となることが想像できるだろう．

我々はこのような，空の状態で作られ，後から 1 文字ずつ後ろに文字を足していくことができるような部品を，さらなる下請け部品 (tokenizer から見れば孫請け部品) として作ることにしよう．

このような「部品の細分化」は最初から完全な見通しのもとで行えるものではないので，柔軟に新しい部品を導入する，部品と部品の境界を「美しい方向へ」(決して汚い方向へではなく) 変更していくことが必要である．よく,「モジュールの仕様書や設計書を書いてからコードを書く」などと理想論を語る人がいるが，あまり信用しないほうが良い．設計書がタコければ，それを実装する人は，その制約に縛られてひどいコードを書かされるだけである．コードを書きながら自分で部品と部品の境界線を美しく整えて行ける人が一番早く，動くコードを書ける．ただし，多人数でのプロジェクトでは，部品間の境界の変更はおおごとになるので，あまり気楽にやれないこともある．本演習ではそのようなことはないはずである．

話を元に戻して，もうひとつ面倒なのが，indent/dedent 字句を正しく生成する部分である．基本的な方針は，前掲した Language Reference 中の記述に従うことである．

そのために，記述中で言及されている stack (int 型のデータを push/pop できる) を，独立した部品として作ることにする．

その上で，実際に上の tok_next という関数のどの辺を変更すればよいのかと言うと,「空白を読み飛ばす」という部分である．具体的には以下のようになるだろう．

- 空白を読み飛ばした際に，それが行の先頭から始まる空白であったならば，そこで読み飛ばされた数を数え，それを「現在行の先頭の空白の数」として覚えておく．
- the stack の top (最後に詰まれた要素) の値 (l) と，現在行の先頭の空白の数 (n) を比べる．

- $l < n$ ならば，indent 字句を生成 (つまり，tok を TOK_INDENT にセット) するとともに，n を stack へ push
- $l = n$ ならば何もせずに通常の処理をする．
- $l > n$ ならば，stack から 1 要素を pop して．dedent 字句を生成する．このとき，1 要素 pop した後で stack top の値 $< n$ となってしまったらそれは正しくない字下げである (“it must be one of the numbers occurring on the stack” に反する) ので，エラーを表示する．たとえば以下のような場合におきる．

```
if x < 2:
    return 1
  else:   <--- この行でエラー発生
    return ...
```

かくして，我々の tokenizer を作るための下請け部品として以下のようなものを作ることになった．

- char_buf_t: 文字のバッファ．文字の追加および追加された文字を文字列として取り出す機能を持つ．をためておく．
- int_stack_t indents: 整数のスタック．push/pop ができる．
- char_stream_t: ファイルから文字を読み出していくとともに，現在行や現在位置 (ファイル名，行番号，列番号) を把握する．

以下では，これらの部品を順に説明した上で，字句解析器の仕組みの概要を説明する．

4.2 部品: char_buf

- char_buf_t mk_char_buf()

  空の char_buf を作る．

- int char_buf_reset(char_buf_t b)

  b を空にする．

- void char_buf_append(char_buf_t b, int c)

  b に文字 c を加える．

- int char_buf_size(char_buf_t b)

  最後に reset されてから (一度もされていない場合は作られてから)，加えられた文字数．

- char * char_buf_get_string(char_buf_t b)

  最後に reset されてから (一度もされていない場合は作られてから)，加えられた文字を順に持つ文字列を返す．

という単純なもので，Java の StringBuffer のようなものである．

また，C++で，STL (標準テンプレートライブラリ) を使いこなしている人は，その中に，上の機能をすでに持っているようなライブラリがあることも知っているだろう．本当に馬鹿馬鹿しくてやってられないという人は無理に作る必要はないが，そうでない人は，単に部品としてあるから使う，というのではなくできれば実際に作ってみてほしい．

作り方は簡単で，基本的には内部に，malloc により確保された配列 a とその大きさ，さらに，現在 a に格納されている文字数を持っておき，a に文字をためていけばよい．ただし，配列 a の大きさが足りなくなる場合がある．そのときは，a より大きなサイズ (たとえば 2 倍に) の領域を新たに割り当てた上で，古いデータをコピーする．このように単なる配列と違い，動的にデータを増やしていけるところがポイントで，このようなデータ構造は，部品化しておく価値がある程度には複雑なものである．

余計なおせっかいだが，char_buf_get_string は，配列 a をそのまま返すだけではいけない．理由は，a がその後で変更されるかもしれないからである．したがって「その時点での内容」をコピーして返す必要がある．malloc をして，コピーをして，返しても良いし，strdup という「文字列コピー」ライブラリを使っても良い．しかしどちらにしても重大な注意は，

> 文字列は最後が 0 (文字の'0' ではなく，整数の 0) で終わっていなくてはならない

ということである．単に配列 a に文字をためて行っただけでは，n バイト目まで書いた状態でも，$n+1$ が 0 という保証はない．したがって単に，strdup(a) としただけでは，何バイトコピーされるのかは運任せである．strdup を呼ぶ直前に，$a[n] = 0$ を行うか，append した直後 (および作られた直後) に常に $a[n] = 0$ を実行しておくのが良い (後者の方がなんとなく安心感がある)．この際，配列 a の大きさを間違えずにとっておくことが必要である ($a[n] = 0$ を実行するには，a が $n+1$ バイトないといけない)．

4.3 部品: int_stack

- int_stack_t mk_int_stack()

  空の int_stack を作る．

- void int_stack_push(int_stack_t s, int c)

  s に文字 c を push する．c が stack の top の要素となる．

- void int_stack_peek(int_stack_t s)

  s の top の要素を取り除かずに返す．

- void int_stack_pop(int_stack_t s)

  s の top の要素を取り除いて返す．

さしたる説明は必要ないであろう．これも C++の STL に似た部品がすでにある．

空の stack に int_stack_pop を呼ぶのはもちろんあってはならないが，部品のコードを書く際は，必ず空でないことを検査をした上で，違反が起きていたら，エラーを出してプログラムを終了させ

るように書いておくことが必要である．それを怠ると，意味のないデータが読み出され，プログラムの挙動は意味不明になり，デバッグがしにくくなる．

4.4 部品: 拡張された char_stream

先週のウォームアップで作った部品を，tokenizer のために若干拡張する．また以下では，先週の説明で char_stream_next_char などと読んでいた関数を，cs_next_char などと呼びかえている (短くするため)．

- char_stream_t mk_char_stream(char * *filename*) char_stream を作る．
- void cs_next_char(char_stream_t *cs*) 次の一文字を読む．
- int cs_cur_char(char_stream_t *cs*) 現在の文字 (最後に cs_next_char が呼ばれたときに読まれた文字) を返す．
- char * cs_filename(char_stream_t *cs*) 読み込み中のファイル名を返す．
- int cs_cur_line_no(char_stream_t *cs*) 現在の行番号を返す．
- int cs_cur_column_no(char_stream_t *cs*) 現在の列番号を返す．
- char * cs_cur_line_string(char_stream_t *cs*) 現在の行の先頭から，現在の文字までの文字列を返す．

cs_cur_line を実現するために，char_buf 部品を使う．

4.5 あらためて tokenizer と，tok_next の概要

以上を踏まえたうえで改めて，tokenizer と tok_next の動作の概要を示す．まず tokenizer は内部的な要素として，少なくとも以下を持つ必要があるだろう．[1]

```
typedef struct tokenizer
{
  char_stream_t cs; /* ここから文字を読み出す */
  char_buf_t tok_buf; /* 現在構築中の字句を構成する文字を保持 */
  char_buf_t str_buf; /* 現在読み込み中の行を構成する文字を保持 */
  int_vec_t indents; /* 字下げスタック */
  int leading_spaces; /* 現在の行の先頭の空白の数 */
  token tok; /* 現在の字句 */
} tokenizer, * tokenizer_t;
```

[1]以下を丸写しする必要はない．各自必要だと思ったものは付け加えていくこと．

各々が必要な理由は，上の説明とコメントを見れば明らかだろう．
その元で，tok_next は以下のようなことをすることになるだろう．

```
void tok_next(tokenzier_t t)
{
  /* 現在の字句をためておくバッファを空にする */
  char_buf_reset(t->tok_buf);
  /* 空白読み飛ばし */
  skip_whitespaces(t);
  /* indent/dedent 字句を生成する必要があればそれで終わり */
  if (t->leading_spaces > int_stack_peek(t->indents)) {
    int_stack_push(t->indents, t->leading_spaces);
    t->tok.k = TOK_INDENT;
    return;
  } else if (t->leading_spaces < int_stack_peek(t->indents)) {
    int_stack_pop_last(t->indents);
    if (t->leading_spaces > int_stack_peek(t->indents)) {
      err_wrong_indent(t);
    }
    t->tok.k = TOK_DEDENT;
    return 0;
  }
  /* ここからがいわば本題．文字を読んでいきながら場合わけ */
  switch(cs_cur_char(t->cs)) {
    case ...
    case ...
    case ...
    ...
  }
}
```

skip_whitespaces(t) の詳細は各自考えよ．注意としては，

- 空白だけでなく，コメントも読み飛ばすこと．
- 前述したとおり，行全体が空白 + コメントだけであった場合，その行からは字句は生成されない．そうでない場合は，行の最後の改行字句が生成される (コメントがあろうとなかろうと)．
- skip_whitespaces(t) が呼ばれた時点で，現在の文字が行の先頭であった場合，読み飛ばされた空白の数を t->leading_spaces に正しくセットすること．この要素は常に,「現在の行の先頭の空白の個数」を保持するようにする．

などがある．文字通り空白を読み飛ばすだけではいけない．

さて，上のコードの大きな switch 文の中身は各自に書いてもらうことになるが，ひとつだけ例を書いておこう．たとえば，cs_cur_char(t->cs) が数字であった場合，それは数の始まりである．したがって以下のような感じになるだろう．

```
  case '0'...'9':
    scan_number(t);
    break;
```

そして scan_number の役割は，その数の終わりまで文字を読んでいくすなわち scan_number が終わった時点で，cs_cur_char(t->cs) は，その数に入らない先頭の文字を返すようにする．たとえば，358*4 のような文字列で，cs_cur_char(t->cs) が先頭の'3' である状態で scan_number を呼ぶと，cs_cur_char(t->cs) が'*' となる状態まで文字を読み込んで復帰する．当たり前のことだがこういうのも，各関数の動作を一度明文化して，似たような関数の間 (たとえば他にも scan_identifier のような関数ができることだろう) で統一された慣習を守るように心がけることで，プログラムのバグが少なくなるのである．

さて，scan_number の中身だが，闇雲にコードを書き始める前にきちんと，認識すべき文字列の定義を書いておこう．これはコメントとして，関数の直前などに書いておくと良いだろう．

```
/*
  integer = 0 | non_zero_digit digit*
  float   = digit* [. digit* ]
  non_zero_digit = (1|...|9)
  digit          = (0|1|...|9)

  現在の cs_cur_char(t->cs) を含み，上記のパターンに合致する文字列までを読んで
  行く．復帰時には，cs_cur_char(t->cs) は上記パタンに合致しない最初の文字になっ
  ている．
*/
void scan_number(tokenizer_t t)
{
  ...
}
```

肝心の本体だが，上の定義から形式的に導くよりも，integer もいったん，digit*とみなした上で，もし 0123 のような文字列が読み込まれていたらそれを後から拒否することにしよう．

```
  integer = digit*  # ただし 0 から始まった場合は``0'' でなくてはならない
  float   = digit* [. digit* ]
```

すると全体は以下のようになるだろう．

```
void scan_number(tokenizer_t t) {
  scan_digits(t);
```

```
  if (cs_cur_char(t->cs) == '.') {
    /* 浮動小数点数 */
    eat_char(t);
    scan_digits(t);
    t->tok.k = TOK_LITERAL_FLOAT;
    t->tok.v.v_float = atof(char_buf_get_string(t->tok_buf));
  } else {
    /* 整数. 0123 などを排除 */
    char * s = char_buf_get_string(t->tok_buf);
    if (s[0] == '0' && strlen(s) > 1) {
      err_wrong_token(t);
    }
    t->tok.k = TOK_LITERAL_INT;
    t->tok.v.v_int = atoi(char_buf_get_string(t->tok_buf));
  }
}
```

eat_char(t) は現在の文字を tok_buf に格納しつつ，次の文字を読む．scan_digits は，名前が示すとおり，数字以外が現れるまで文字を eat_char していくものである．

整数リテラルの認識はそらく tokenizer の中では複雑な部類に属し，その他の大部分はもっと易しいはずである (これよりも若干面倒なのは，文字列リテラルの処理くらいだろう)．

その他，注意が必要な点は，for, def などの，アルファベットからなる予約語は種類が多いのでまともに,「case 'f':」などと場合わけをしていくと大変である．いったん全部を identifier として認識して，その後改めてその identifier をすべてのアルファベットからなる予約語と，strcmp を用いて比較するほうが単純であろう (若干遅くなるが，気にするほどのことはない)．

5 メモリ割り当て・管理について

5.1 C 言語でのメモリ割り当て

C のプログラムでメモリを確保するには，大別して，

- 大域変数・大域配列として確保する (文法上は，関数の外側で定義した変数や配列)
- auto 変数・auto 配列として確保する (文法上は，関数の内側で定義した，ローカル変数・配列)
- 一般的なヒープから確保する (malloc を用いて領域を割り当てる)

がある．C++の場合，最後が new になるだけである (new も malloc も仕組みは同じと思ってよい)．

適材を適所で使うとしか言いようがないが，auto 変数・auto 配列は,「それが現れている関数の呼び出し中しか有効でない」ことにくれぐれも注意が必要である．したがって，関数の復帰後も有効であるようなデータ構造を割り当てるには，決して使ってはいけない．たとえば以下のようなプログラムは書いてはいけない．

```
char_stream_t mk_char_stream()
{
  char_stream cs[1];
  cs->... = ...;
  cs->... = ...;
  return cs;
}
```

というのは，mk_char_stream で auto 配列として確保された配列 cs[1] は，mk_char_stream が復帰した後はもう存在しない (もちろん実際には存在はするが，他の関数呼び出しの際に同じ領域が使われて知らぬ間に書きつぶされていく) ためである．

C 言語の恐ろしい原則に従い，コンパイラもこれをコンパイル時のエラーとはしない．gcc で -Wall をつけてコンパイルすると，警告は出るが，それもプログラムの書き方の詳細に依存する．警告が出なくなることもある．放っておいて実行したときの動作も文字通り不確定で，まったく関係ない代入をした瞬間に知らぬ間にデータが書き換わったり，逆に cs の要素を書き換えたときに他の関係ないデータが書き換わったりするようになる．しばらく走り続けた後で，関係ないところで突如不正メモリアクセスを起こして終了したりと，ありとあらゆる動作が可能になる．

大域変数にはこのような問題はない．したがって上記のプログラムを

```
char_stream cs[1];
char_stream_t mk_char_stream()
{
  cs->... = ...;
  cs->... = ...;
  return cs;
}
```

と書き換えれば一見問題はなくなる．しかし，大域変数は文字通り大域的で，どこでも同じ領域をさすわけだから，世の中に char_stream_t の構造体は一個しか作れないということになる．これは，mk_char_stream という関数の名前にふさわしい動作とはいえないだろう．

結局このような関数を書くために使える方法は，3 番目の malloc (または C++ の new) しかないことになる．

```
char_stream_t mk_char_stream()
{
  char_stream_t cs = (char_stream_t)malloc(sizeof(char_stream));
  if (cs == 0) { /* memory allocation failed */
    ...; /* show error msg */
    exit(1);
  }
  cs->... = ...;
  cs->... = ...;
  return cs;
```

```
}
```

とすればよい．このようにありとあらゆるところで malloc が使われることになるだろう．これまでに述べた関数でも，各種の mk_XXX 関数が malloc を使うだろうし，それ以外にも，char_buf_append や int_stack_push などでも使うことになるだろう．

5.2 メモリの解放

auto 変数・配列はそれを定義した関数呼び出しが終了したときに自動的に解放される．大域変数は，決して (というより，プログラム終了まで) 解放されないが，プログラム中に増えるものではないので，あまり解放する必要がない．しかし malloc された領域は,「必要に応じて」解放する必要がある．それには free という関数を呼べばよいのだが，その呼び時を間違えて，まだ使っているデータに対して free を呼ぶと，先に示したような，間違えて auto 変数を用いた場合と同じ問題を食らうことになる．

我々が作る部品も，mk_XXX があればそれに対応する free_XXX があってしかるべきだが，実際にはその free_XXX をどこで呼ぶかが大問題となる．Java をはじめとする (実は C/C++/Fortran 以外のほとんどの) 言語には，自動ごみ集め (Garbage Collection) という機能があり，プログラマがこの問題で頭を悩ませる心配はない．我々のアプローチは以下とする．

- 当面 free のことは忘れる．malloc (new) するだけで決して free (delete) をしない．
- そのうち，C/C++でも使えるごみ集めライブラリを紹介する．
- オプション課題として，自前でごみ集めを実装する．

もちろんこれがいやな人は慎重を期して free できるデータを free してみよう．ただし我々は言語処理系を作っているので，一般的なごみ集めを作ることなしにきちんとメモリ管理をすることは最初からほとんど不可能であることに注意しておこう (つまりはその苦労はあまり意味がない).

6 まめ知識 : デバッガ

6.1 デバッガの一般的な機能

ブレークポイント: プログラムの実行を途中で止める．止める場所の指定の仕方としては，関数名，ファイルの行番号などがある．

実行: プログラムを，ブレークポイントに達するまで実行する．

ステップ実行: プログラムがブレークポイントで止まっているときに，次の文まで実行してまた止めたり，次の行まで実行してまた止める，など．指定の仕方としては，関数名，ファイルの行番号などがある．

スタックトレース表示: プログラムがブレークポイントで止まっているとき，その関数呼び出し履歴 (main 関数からどのような呼び出しを経て現在位置にいるか) を表示する．

変数や式の値表示: プログラムがブレークポイントで止まっているとき，とまっている場所で有効なローカル変数や大域変数の値を表示する．

フレーム間の移動 スタックトレースをさかのぼって，現在の関数を呼び出した関数 (や，さらにそれを呼び出した関数，etc.) に移動し，そこで有効なローカル変数の値を表示させたりすることができる．

以上が非常に良く使うデバッガの機能である．これは gdb でも Microsoft の IDE に組み込まれているデバッガでも，概念としてはほぼ同じといってよい．

全体として，デバッガは一言で言えば，プログラムの実行を inspect するための道具である．通常プログラムの挙動は，それ自身が print などを使って出力したものを通してのみ人間に観測できるわけだが，デバッガでその中身を見ることができるようになるというわけである．そのために適切な場所でプログラムの実行を止めたり，ステップ実行によって実行の様子を追いかけたりする．

6.2 Emacs におけるデバッガの立ち上げ方

先週の繰り返しになるが,

- -g オプションをつけてプログラムをコンパイルする
- バッファにソースコードを開いた状態 (正確には，実行可能ファイルと同じファイルであればなんでも良い) で,

  `M-x gdb`

  を実行.

  `Run gdb (like this) : gdb`

  に対して，後ろに実行可能ファイル名を追加して，リターン．デバッガが Emacs のバッファ内で立ち上がる．後は通常の gdb と同様であるが，ブレークポイントなどでプログラムが止まるたびに，ソースファイルの該当場所を自動的に表示してくれる (これは gdb を単独で起動した場合にはない機能で，Emacs から使う場合にのみ得られるメリットである．実際にはこれなしでデバッガを使う気にはなれないというほどの違いをもたらす).

6.3 gdb のコマンド

マニュアルは，Emacs 内で,

`M-x info`

とするといろいろなコマンドのマニュアルが参照できる．このうちの，gdb のマニュアルを時間のあるときに読んでみることを進める．

以下のコマンドは，gdb のプロンプト (`(gdb)` という文字列) に対して入力する．一部は Emacs のキーにバインドされている．

ブレークポイント (break/b): コマンド名は break で，省略形として，b だけでもよい．以下，上記のようにカッコ内にコマンド名，スラッシュ後にその省略形を書く．

- b 関数名
  で，指定された関数が呼び出された際に停止する．例:

  ```
  b main
  ```

- b ファイル名:行番号
  で，指定されたファイル名の指定された行番号に実行が差し掛かったとき (実行される前に) 停止する．例:

  ```
  b char_stream.c:157
  ```

  ただし，Emacs ではこうする代わりに，ソースコード上の止めたい行にカーソルを置いた上で，

  ```
  C-x SPACE
  ```

  で，同じことができる．これも，Emacs 内でのみ得られる強力なメリットである．

実行 (run/r, continue/c): run と continue というコマンドある．

- r 引数...
  で，プログラムを先頭から，引数を与えて実行する．
- c
  で，プログラムを，現在止まっている位置から続けて実行する．

どちらも次のブレークポイントまで実行する．また，プログラムが不正メモリアクセスなどのエラーを食らったときもブレークポイント同様，止まった位置が表示される．

ステップ実行 (next/n, step/s): next と step というコマンドある．

- s
  現在の行と違う行に差し掛かるまで実行する．
- n
  現在の関数呼び出し内で，現在の行と違う行に差し掛かるまで実行する．s との違いは，現在の行が関数呼び出しを含んでいた場合，その関数呼び出し内では止まらずに実行が続くことである．

たとえば現在の行が

```
f(x);   <----- 現在の行
g(x);
```

となっていたら，s は f 内部の 1 行目で止まり，n は，g(x) の行で止まる．

要するに関数の内部に突入していくのが s, 関数呼び出しを乗り越えるのが n である．実際 Microsoft のデバッガなどでは，前者を step into, 後者を step over などと呼んでいる．

スタックトレース表示 (where/bt): where または bt というコマンドある．動作は同じである．

- `bt`
  としてみれば，関数呼び出し履歴が表示されるのでやってみよう．

変数や式の値表示 (print/p, display/disp): print および disp というコマンドがある．

- `p` 式
  で，式の値を表示する．式は，現在止まっている場所で有効なローカル変数を含んでよい．
- `disp` 式
  は p と似ているが，ブレークポイントで実行が止まるたびに (その式がその場所で妥当であれば)，自動的に値が表示される．実行につれて変数がどのように変わっていくかを見たいときに便利である．

フレーム間の移動 (up/down) up および down というコマンドある．

- `up`
  で，注目するスタックフレームを一段あがる (呼び出し側に移動する)．
- `down`
  で，注目するスタックフレームを一段下がる (呼び出された側に移動する)．

たとえば，`main` → `f` → `g` という呼び出しが行われていて，現在 `g` で停止しているとする．この場所では `g` 内のローカル変数の値などを見ることができるが，このままでは `f` や `main` のそれを見ることはできない．

ここで，up コマンドを実行すると，注目のフレーム (関数呼び出し) がひとつ上に移動し，`f` になる．こうすると，p コマンドで `f` のローカル変数などを参照できるようになる．

6.4 デバッガを使えば一瞬で見つかるバグの例

以下のようなプログラムをわざと書いて，実行してみよう．

```
main()
{
  *((char *)0) = 0;
}
```

Segmentation fault というメッセージとともにプログラムが終了するはずである (Windows では例のおかしな日本語のダイアログボックス[2]が出てくるが意味は同じである)．-g オプションをつけてコンパイルし，デバッガで実行してみよう．

すると，デバッガが

```
*((char *)0) = 0;
```

の行でプログラムが Segmentation fault を起こしたことを教えてくれる．

詳しくは OS の授業で出てくるが，Segmentation fault は，要するに「不正なメモリアドレスにアクセスした」ということである．この例ではわざと 0 番地をアクセスしているが，C 言語ではポインタの初期化を忘れたり，配列の添え字を間違えたりして，本来アクセスしてはいけないアドレスにアクセスすることは簡単に起きる．

この例はもちろん，そもそもデバッガを使わなくても，誰でもわかるバグであるが，上記のようなミス (ポインタの初期化忘れなど) が大きなプログラムの中の一箇所で起きている際には，デバッガの威力が発揮されるだろう．たとえば，プログラムが不正メモリアクセスで落ちた場合，以下のような手順で問題がすぐに解決する場合は結構あるものである．

- プログラムをデバッガ内で走らせると，プログラムは不正メモリアクセスを起こした場所で止まりデバッガがそのソースコード上での場所を表示してくれる．
- その場所では必ずメモリアクセスが行われているはずである．ほとんどの場合，その場所に `*a`, `a[i]`, `a->x` などの，ポインタを通したメモリへのアクセスが行われていることだろう．
- そしてこのような場合，多くが `a` というポインタ変数におかしな値が入っている (最も単純なケースは，初期化し忘れた結果，偶然 0 が入ってたというケース) か，配列の添え字 `i` がおかしな値 (ありえないほど巨大だったり，負だったり) が入っていたりするはずである．これは，デバッガの p コマンドなどを使えば簡単に調べることができる．
- **そのプログラム周辺を見て**，`a` への代入を忘れていたり，添え字 `i` への代入を間違えていたり，ループを止める条件を間違えて `i` が巨大になるのが放置されていたり，などの明らかな間違いがすぐに見つかれば幸運である．

4 つ目の要素は，ある種の楽観論であり，いつも「プログラムが最終的に異常な動作 (例: 不正メモリアクセス) をしたとき」と「その原因となる間違い」がソースコード上で近くにあるとは限らない．しかし，なにしろ C 言語はポインタの初期化のし忘れや配列の添え字オーバーフローなどの初歩的な間違いを犯しやすい言語だから，逆に言うと最初に見つかるバグが，上のような手順で一瞬で解決することは多いのである．

とくに，不正メモリアクセスのようなバグは，デバッガを使わない限り，プログラム自身が有用な情報を残さずに終了してしまうため，デバッガの威力が発揮される．

[2] アドレス 0x.... が “read” になることができませんでした

7　デバッグの一般論

デバッガと言うものが，プログラムを与えると勝手にソースコードの間違いの場所を指摘してくれるようなものであれば楽なのだが，今見たように，デバッガ自体は単なるプログラムの実行を inspect できる道具に過ぎない．

本当は，デバッ「ガ」の前にデバッ「グ」の話をすべきである．デバッガを使うことはデバッグのごく一部でしかない．たとえば世の中には通称 “printf デバッグ” と呼ばれるような原始的だが実際にはよく使われ，役に立つ方法もある．ここでは，デバッグをする際の心構えや，デバッグをする際に，何を方針にして「次の一手」を選んでいくべきかについて，一般論は無理なことを承知で，しかし有用なガイドラインらしきものを述べてみよう．

7.1　一番重要な原則

意外に聞こえるかもしれないが，デバッグをする際の一番重要な原則，心構えは，

あまりプログラムを直そう直そうと思ってはいけない

ということではないかと思う．デバッグをする目的は，最終的にはプログラムを直すことなのだから,「プログラムを直そうと思うな」とは，目的とまるで逆さまのことを言っているようだが，どういうことなのか少し説明をしてみる．

デバッグをあまり経験したことのない初心者はプログラムの挙動不審に対して，自分のソースコードを端から端までじっと眺めるということをはじめてしまうことが多い．こうしてしまう理由のいくつかを想像してみると

- それしか方法を知らないから
- 小さなプログラムの場合，それが最も早いことも多いのでその考え方を延長して
- 多少プログラムが大きくなっても,「間違う心当たりのある場所」がなんとなく見当がつく場合もあり，運がよければそこですぐにバグが見つかるので，それを期待して

などがあると思うが，プログラムが大きくなってくると通用しない場合も多い．そのような場合には,「間違い発見のためのソースコード眺め」は砂漠で針を見つけるがごとき作業で，やっているほうも見通しなくやっている場合が多い．そして，怪しげなコードを見つけては修正をしてみて，無駄な時間を費やす可能性もある．これは精神衛生上も非常に悪い．しまいには「なぜ動かないんだ!」となかばキレタ状態になってキーボードをたたき出す，などと言うことにもなる．

それだけならよいが，もっと悪いのは，そういうことを繰り返していくうちに，関係のない修正を施した結果，一見バグがなくなったかのような挙動をプログラムが示してしまう場合である．これは，本当のバグを除去したわけでもないのに，それを隠してしまったことになる．そして後に別のデータでプログラムを走らせたり，プログラムをさらに修正していったとき，そのバグが再び顔を出すことになる．こうなってくるともう，発狂する以外に逃げ道がなくなってくる．

心当たりをあたってすぐにバグが見つからなければ,「プログラムを動くように直したい」という気持ちを一度捨て去ることが重要である．

7.2 正しい「心構え」

ではそのような場合の正しい心構えは何なのか．それは，

デバッグ = 現在のプログラムの挙動調査・診断

という気持ちである．プログラムが正しいかどうかを知っているのは人間だけで，コンピュータは正しいプログラムも間違ったプログラムも同じように実行しているだけである．現在の間違ったプログラムがなぜそのような挙動をするのかを「調査する」というある種の第 3 者的な気の持ちようがうまくいく.「おやまあ，どれどれ，君はなぜそんな動作をするのかね」という心境 (平常心) が重要である．決して「なぜ貴様は動かないんだ!!」ではなく．チームでプログラムを組んでいるなら，一人がコーディングをしたプログラムを，もう一人がデバッグするというのも良いだろう．

この調査・診断についてもう少し詳しく述べよう．

おそらく現在目の前にあるプログラムも，main 関数の 1 行目から間違っているわけではなかろう．つまり一般に「途中まではうまく行っていた」プログラムが，最後には正しい実行の軌道を外れて別の道を歩んでいることになる．

デバッグとはこの,「正しい挙動」と「自分のプログラムの挙動」が分岐する，分岐点を見つけること

ここでいう挙動の「分岐点」とは自分のプログラムが異常な出力や，不正メモリアクセスで落ちるなどの，明らさまに異常な挙動をした地点をさすのではなく，本来の挙動から逸脱した最も早い地点をさす．たとえばある場所で変数 x の値を間違えてセットしたとする．この場合，その場所が分岐点である．しかし，プログラムのあからさまな異常動作はもっと後になって現れるかもしれない．たとえば，しばらくたってから x の値が偶然出力されて，間違いがあることに気づくときもあるし，しばらくたって x をポインタとしてメモリをアクセスしたら不正メモリアクセスで落ちて，それに気づくときもある．x が別の変数へ代入されて間違いが各所に「伝播」していって，そのうちにそれらの変数のどれかが悪さをすることもあるだろう．したがって，プログラムがあからさまに異常な動作 (異常終了や異常な出力) をした地点と，求める分岐点 (バグのある場所) は，一般には遠く離れている可能性がある．したがってあからさまに異常な動作をした地点から，実際の分岐点を見つけていく過程こそがデバッグである．

デバッグを始める時点で分かっている情報 (あるいは簡単に分かる情報) は，

- プログラム開始時点では，プログラムはまだ正しい軌道の上にいる
- あからさまな異常が観測された時点で，プログラムはすでに正しい軌道からはずれている

という 2 点である．

システマティックなデバッグの過程とは，この状態からある種の「2 分探索」をはじめて，

- ここまでは正しい軌道上にいる (らしい)
- ここではすでに正しい軌道を外れている

図 1: デバッグのイメージ図．点線はプログラマの頭にある正しいプログラムの軌道．実践は現在のプログラムの挙動．どんなプログラムも main 開始時は正しい軌道上にいる．プログラムがあからさまに間違った動作 (異常終了や異常な出力) をしたときは明らかに，間違った軌道上にいる．難解なバグをデバッグするシステマティックな方法とは，プログラムが正しい軌道を逸脱した地点 P を探索していくことである．それには，ここまでは正しい地点，ここではすでに間違っている地点の間を徐々に狭めていけばよい．そのために，デバッガのブレークポイントと inspection 機能を使ったり，printf を挿入して情報を収集する．

という,「分岐点の存在する範囲」を徐々に狭めていくという過程のことである．

机上の理論ではあるが，以下のような過程でデバッグをしていけば，どんなバグも必ず見つかることになる．[3]これは机上の理論だが，実際に難解なバグを追い詰めていく際のプログラマの頭の中にある思考回路の表現としては，よい近似になっていると思う．

```
C = プログラム先頭地点; /* Correct point */
W = 異常動作発覚 (異常終了，異常出力，etc.) 地点; /* Wrong point */
while (! C と W がある程度遠い) {
  M = C と W の大体の中間地点;
  M 地点でのプログラムの状態を調べ，プログラムが「まだ正しい軌道上にいる」
   か否かを調べる．
  if (M 地点でプログラムは正しそうだ) {
    C = M; /* 分岐点は M 以降にある */
  } else {
```

[3]ただし，同じプログラムを何度走らせても常に同じ挙動をする (決定論的動作をする) ことが前提である．非決定的な乱数，スレッド，ネットワークなどを使うプログラムでは成り立たないこともある

```
    W = M; /* 分岐点はM以前にある */
  }
}
```

ここで,「C と W の大体の中間地点を見つけてその場でプログラムの状態を観察する」という作業をするのに，あるときはデバッガのブレークポイント機能などを使うし，あるときはプログラム中に print 文を挿入して状態を出力させる，など，様々な方法が使われる．

上のプロセスを実現するために実際に行うことと，その際のいくつかのコツを述べてみよう．

プログラムの入力・実行時間を極力小さくする: 分岐点を見つけていくのが目的だから，探索範囲は最初から小さいほうが良い．同じバグが発生する範囲で，プログラムの入力 (言語処理系であれば，実行させるプログラム) を極力小さくしたり，実行時間を極力小さくするというのは，言うまでもない原則である．少し難解なバグを，本腰入れて突き止める場合，必ず行うステップといってよい．

実際にはここで非常に小さな入力でバグが発生することが分かると，システマティックな 2 分探索を始めるまでもなく，バグのありかが想像できてしまう場合も多い．

デバッガで，おかしな挙動をした周辺のプログラムを眺める: 6.4 節で「デバッガを使えば一瞬で見つかるバグ」の例を説明したが，これはその一般化といえる．

つまり，プログラムがあからさまに間違った挙動 (e.g., 異常終了や異常な出力) を示した地点 (上記の W 地点) の比較的すぐ近くにプログラムのバグがあることが結構多いということである．そのような場合，プログラムを地点 W の周辺で (ブレークポイントを用いて) 停止させ，どの変数の値が間違っているかなどを (デバッガの p コマンドなどで) 調べ，その変数がどこで代入されたかなどを，周辺のプログラムを眺めて調べる．そのような,「W 周辺の局所探索」をすることでバグがすぐに見つかってしまう場合も結構多いということである．

いわゆる printf デバッグ: 分岐点を 2 分探索で見つけるといっても，2 分探索で探索範囲を狭めるために必要な作業はそれなりに多くの手作業が必要になる．デバッガは便利だが，一時には一箇所でのプログラムの状態を調べることができるに過ぎない．

printf デバッグは，デバッガがなくても使える古典的な方法だが，これも結局,「分岐点探し」の 1 方法と位置づけることができる．つまり，適切な場所で適切な変数の値を出力させれば，それを眺めることで,「どこまでは正しい出力で」「どこからが間違った出力か」が，わりと狭い範囲に特定できるのである．これはデバッガで文字通り一連の作業で探索範囲を半分に狭めていくのと比べると，実際には非常に効率的なことが多い．

printf デバッグをする際には，闇雲に変数の値を表示させるのではなく,「正しい出力が何であるか」をきちんと意識して，print 文を挿入することが重要である．すなわち，実際の出力を見れば合っているか間違っているかどうかが判定できることが重要である．

この演習中に，闇雲ではないデバッグ，システマティックなデバッグ，というものを，意識して行えるようになってくれることを期待する．