B演習 (言語処理系演習) 資料 (2)
小さな部品の開発，CVS を用いたソースの共有

田浦

2006/10/23

1　言語処理系というソフトウェアの概要

1.1　構文木，構文解析

言語処理系をもっとも大雑把に眺めるならば，それは「プログラム」である文字列を入力し，その文字列に書かれていることを実行するソフトウェアである.「実行する」ことによって，そのプログラムが起こすべき副作用 (画面への表示，ファイルへの出力など) がおき，それを見て正しくプログラムが実行された，と思うわけである.

プログラム (文字列) → 処理系 → そのプログラムの出力・副作用

ほとんど情報量がないが，あえて擬似コードで書けば,

```
mini_python(file)
{
  char * program_string = open_and_read_file(file);
  exec_program_string(program_string);
}
```

くらいになる (これはあくまでこの場での説明用である．いきなりこれを字面どおり真似してプログラムの作成に取り掛かるようなことはしない).

ここで処理系によって，プログラムを与えてそれを実行するまでをひとつのコマンドとして実行するものと，与えられたプログラムをいったん機械語，またはその他の言語に変換し，それを改めて実行する，という手続きを踏むものとがある．www.python.org からダウンロードできる python 処理系は前者に属する．後者の例としては gcc などの C 言語の処理系，また Sun が提供している Java の処理系などがある．たとえば C/C++言語の処理系はほとんどの場合，入力されたプログラムを，オペレーティングシステムと CPU によって直接実行できるフォーマットに翻訳する．Java の場合，javac というコマンドによって Java プログラムが，クラスファイルと言う，ソースファイルでも機械語でもない形式に変換され，それが java というコマンド (Java Virtual Machine) によって実行される.

しかしながら前者に属する処理系も，処理系の内部で (C の処理系のように) 機械語を生成して実行したり，その他の中間言語に変換してから実行したりするものが多いので，両者を厳密に線引きすることに大した意味はない．本演習で解説する処理系は前者に属している．

さて，そのような処理系の内部をもう一段詳しく見ると以下のようになる．

プログラム (文字列) → 構文解析器 → 構文木 → 評価 (実行) 器 → そのプログラムの出力・副作用

擬似コードで表現すると，

```
mini_python(file)
{
  char * program_string = open_and_read_file(file);
  program_t program = parse_program(program_string);
  exec_program(program);
}
```

くらいになる．

処理系の役目は，文字列を読み込んで，正しくないプログラムは排除し，正しいプログラムは正しく解釈して実行するだけなのだが，それを一度に行うプログラムを作るのは見通しが悪い．たとえば，“1 + 2” を表示するプログラムは，

```
print 1 + 2
print 1+2
print 1   + 2
print    1   + 2
```

などといくらでも表記の自由度がある．これらのプログラム片はすべて，

- print 文であって，
- その引数は，足し算を行う式であって，
- その足し算の引数は，1 と 2 である

という意味で，(構文的に) まったく同じとみなせるものである．プログラムで指定された動作を処理系が実行する際に，このことを毎回毎回，文字列を読み込みながら認識するのは，処理系の見通しも悪くなるし，実行時間の無駄である．もし，上の `print` 文がループの中にあって，100 回実行されたとすると，本来はプログラムを入力した際に一度だけ行えばよい上のような認識を，実行するたびに (つまり合計 100 回) 行うことになる．

したがって**処理系を作る際には，文字列として与えられたプログラムを，後の実行に都合のよいデータ構造に変換するフェーズを最初に設ける**のが普通である．実行に都合の良いデータ構造とは，式や文に対して，その式や文の種類や，それを構成する要素がたちどころに分かるようなデータ構造である．たとえば文であれば，それが print 文なのか，for 文なのか，などの区別が簡単に判定できる要素を持つような，構造体を用いるのがふさわしいであろう．そして，print 文であれば，その引数である式を，やはり構造体の要素として持っている．その他の種類の文に対しても同様に，構

文上の要素を，構造体の要素として持つような自然なデータ構造を作ればよい．言い換えればプログラムの字面上の「構造」(すなわち構文) を素直に表現しているデータ構造である．

たとえば,「while 文」に対応するデータ構造は以下のようになるだろう．

```
/* while CONDITION:
     S1
     S2
     ..
を表現するデータ構造
*/
typedef struct while_statement
{
  expression_t condition;
  statement_vector_t statements;
} while_statement, * while_statement_t;
```

ここで expression_t は「式」を表現するデータ構造で，やはりどこかで typedef によって定義されているはずである．

おそらく，文をあらわすデータ構造は以下のような感じなる．

```
typedef struct statement
{
  statement_kind_t kind; /* for 文なのか，while 文なのか，etc. */
  union {
    ...;
    while_statement_t w;
    ...;
  } u;
} statement, * statement_t;
```

このようなデータ構造を**「構文木」**と呼び，文字列として与えられたプログラムを構文器に変換する (または構文エラーを見つけて排除する) プログラムを**「構文解析器」**と呼ぶ．

かくして処理系内部は文字列を構文木に変換する部分と構文木を実行に移す部分 (評価器 (evaluator)，るいは実行器 (executor)) に分けられた．構文解析器は文字列から構文木への変換を行う「専門家」であり，評価器は，構文木を与えられてそれを実行する専門家である．プログラムを作る際には前者を作る際には後者のことを考えなくてすむように，逆に後者を作る際に前者のことを考えなくてすむように作っていく．

1.2 大きなプログラムの書き方に関する抽象論

プログラム言語処理系の中身がこのように (最低でも) 二つの，ほとんど独立といっていい部分に分かれているのには，上述したような理由 (見通し，速度) がある．実際にはここで用いた考え方

「プログラムをほとんど独立した部品に分ける」は，プログラム言語処理系に限らずほとんどの，中くらい以上のサイズのプログラムを作る際の鉄則である．独立した部品に分ける強い理由があるから分ける，というよりも,「できれば分ける．無理なく分けられるのなら分ける」のが原則だと思っていたほうが良い．

「独立した部品」というのは抽象的な概念だが，目安としては，

- その部品が，それを使うプログラムに提供している「機能」が分かりやすく，単独で説明できる．
- その部品と，それ以外の部分の相互作用が，概念的に少なく，単純である．表面的にはそれは，関数の引数の数が少ない，型の種類が少ない，などの形で現れる．

などの性質を満たすようなプログラムの断片である．当たり前の性質のように見えるかもしれないが，いい加減に関数ごとに分割されたプログラムは，上の性質を満たしていない．たとえば以下のようなことに心当たりはないだろうか?

- 関数に，意味不明な引数が多くある．今たまたまその関数を呼び出している関数の都合で，コンパイルエラーにならないために必要な値がすべて渡されている．
- 関数の返り値の意味が分かりにくい．今たまたまその関数を呼び出している関数の都合に応じて適当な値が返されている．

大きな関数を形式的に分割するとこのような症状になる．関数であれば,「呼び出し側」に言及することなく，その関数自体の意味を単独で，分かりやすく説明できるかどうか，に気を配ると良い．気を配るだけでなく，実際にそれをプログラム中にコメントとして書いておくのが良い．つまり，関数の先頭に,「引数が満たすべき条件，その条件の下でその関数が何をするのか，返り値として何を返すのか」を簡潔に記述するのである．それは説明として「閉じて」いることを目指すべきである．つまり，その関数が呼び出される文脈などに言及することなく，その関数に対する引数だけで説明できなくてはならない (もちろん大域変数に言及しなくてはならないこともあるだろう．しかしその場合もその数を少なくすることを心がけなくてはならない)．そのような閉じた説明が可能でない場合，それは少なくとも部品が提供している機能について，概念 (プログラマの頭の中の大雑把な理解) と実際とで，ギャップがあることを意味する．よく吟味して，概念を実際にあわせるか，実際を概念にあわせるのどちらかをする．前者の場合，その部品の機能が実際にはもっと複雑であることをきちんと認識をする，ということになる．後者の場合，たとえば意味不明な引数がいらなくなるよう，呼び出し側を調整することになる．これが結果的にはよりきれいなプログラムの分割を促進していくことになる．

1.3 言語処理系のさらなる細分化

さて,「部品化」の法則を述べたところで，言語処理系をさらに細分化してみよう．
構文解析器は通常，さらに二つの部分に分割され，以下のように考えることが多い．

プログラム (文字列) → 字句解析器 → 字句列 → 構文解析器 →
構文木 → 評価 (実行) 器 → そのプログラムの出力・副作用

すなわち，字句解析器と構文解析器の二つに分けられる．文字列は文字通り一文字ずつの文字の列である．字句解析器はそれを,「字句 (token)」という，いわば「単語」の列に変換する役割を持つ．つまり，

```
print "hello " + name + " san"
```

という文字列は，プログラムに与えられたときは文字通り，

```
'p', 'r', 'i', 'n', 't', ' ', '"', 'h', 'e', 'l', 'l', 'o', ' ', '"', ' ',
'+', ' ', 'n', 'a', 'm', 'e', ' ', '+', ' ', '"', ' ', 's', 'a', 'n', '"'
```

という 30 文字の文字列であるが，人間にはたちどころに分かるようにそれは，

`print` `"hello "` `+` `name` `+` `" san"`

という 6 つの単語からなる，単語列であるともいえる．前者と後者とを比べると，明らかに後者の方が，構文解析の対象としては単純であろう．

そこで字句解析は，与えられた文字列から，単語 (字句) の切れ目を探し，文字列を字句の列に変換する役割を担う．それと同時にもちろん，そもそも字句として許されない文字列を含むプログラムを排除することも行う．たとえば，

```
print 1a + 2
```

のようなプログラム中に現れる `1a` は，そもそも字句として許されない．

字句解析器は，一つ一つの字句が正しいかどうかを検査するのみで，それ以上の検査は構文解析器が行う．たとえば，

```
for while 1: ...
```

のようなプログラムは，間違った字句を含んでいるわけではないが，プログラム構文としてはもちろん誤りである．

かくして，演習で解説する処理系はおおざっぱに以下の 3 つの部品に分けられた．

字句解析器: 与えられたファイル名らプログラムを文字列として取り出す．その文字列から字句の切れ目を認識し，字句の列に変換する．間違った字句があればその旨を通知してプログラム全体を終了させる．

構文解析器: 字句の列が与えられたとき，それがプログラムの構文として正しいか否かを判定しながら，正しければそれを構文木に変換する．

評価器: 構文木を実行する．

来週以降，この各部品の作り方を説明していく．

2 開発環境について

本演習を実行するための開発環境としては，最低限はエディタ，C (またはC++) コンパイラ，シェルがあればよく，その他にデバッガ，make の使い方を実習する．以下が代表的選択肢だろう．

Unix/Linux: エディタは Emacs を薦める．C/C++コンパイラとして gcc/g++, デバッガとして gdb を使う．複数ファイルからなるプログラムのビルドに make を用いる．授業では，Emacs をいかに快適に「統合開発環境」として用いるかについて，まめ知識集を伝授する (M-x shell, compile, gdb, grep など)．また，Emacs での編集作業を効率化するまめ知識 (M-x query-replace, replace-string, キーボードマクロ) などについても伝授する．

Windows で cygwin: Windows でほぼ Unix-like のコマンド環境が構築できる．したがってエディタ以外は Unix と同様．エディタは Meadow というエディタをインストールすれば，Unix とほとんど同じの環境が構築できる．

Windows で Microsoft Visual Studio IDE: 電気系図書室で CD を借りて Microsoft Visual Studio をインストールすることができる．Visual Studio をインストールすると，統合開発環境 (IDE) を用いてエディタ，コンパイラ，デバッガなどを使うことができる．

Windows で Microsoft Visual Studio コマンドラインコンパイラ: コンパイルは Microsoft のコンパイラで，しかしプログラムの開発は Emacs で，と言う人は，Microsoft のコマンドラインコンパイラ (cl) を用いることもできる．ただし，デバッガとして gdb を用いることはできないので，デバッグ時には結局 IDE を使うことになる．

各種ツール (cygwin, Meadow など) の入手先，Microsoft の開発環境の使い方に関する情報などについては適宜，演習ホームページに情報・情報へのリンクを掲載する．

3 今週の課題

- 開発環境を選択し，使えるようにする．
- 単純ながらも「独立した部品」をを組み合わせてプログラムを作ることを意識してみる．ある機能を提供する部品を作り，それを「使う」プログラムを書いてみる．
- それを二つのファイルに分割してコンパイルしてみる
- Makefile を書いて，コンパイル作業を簡略化してみる
- Emacs とコマンドラインコンパイラ (gcc, Microsoft の cl) の組み合わせを用いてる人は，Emacs を統合環境として用いてみる．Emacs の中からコンパイルコマンドを呼び出す，コマンドを実行する，デバッガを実行する，など．
- CVS を使ってソースをサーバ上で管理する．CVS を使って課題を提出し，チームのメンバ間でソースのやりとりする．

4 プログラムの部品化の練習用課題

4.1 部品の機能

作る部品の機能だが以下のようなものにしよう.

> ファイルを開き，1 文字読み込む.「次の文字を読め」といわれたらさらに 1 文字読み込む．その部品は常に,「最後に読み込んだ文字 (現在の文字)」,「現在の行番号」「現在の列番号 (行の中の何文字目か)」を維持している.

というものである.

そしてその部品を使って，以下の他愛もないプログラム二つを作ってみる.

1. ファイルの行数を数える
2. ファイル中に'a' という文字が現れるたびに，行番号と列番号を表示する.

4.2 部品のインタフェース

その部品は，`char_stream` (文字ストリーム) と呼ばれ，以下のようなインタフェースを持つものとする.

- `typedef char_stream { ... } char_stream, * char_stream_t;`

  `char_stream` は文字ストリームをあらわす構造体で，`char_stream_t` はそれへのポインタである.

- `char_stream_t mk_char_stream(`*filename*`)`

  filename を読み込むような文字ストリームを作る．失敗した場合は 0 を返す.

- `int char_stream_cur_char(char_stream_t` s`)`

  文字ストリーム s の現在の文字 (最後に読み込まれた文字) を返す．ただし，すでにファイルの終端にあるときは特別な値 (EOF) を返す．EOF は，`#include <stdio.h>`をすると自動的に定義される整数値である.

- `int char_stream_cur_line(char_stream_t` s`)`

  文字ストリーム s の現在の行番号を返す．現在の行番号とは，現在の文字が現れている行のことで，最初の文字は 1 行目．'\n' が x 行目に現れると，その次の文字から $x+1$ 行目になる．ただし，EOF だけは特別で，ファイルが空だったときは 0 行目に現れたとみなし，x 行目の'\n' 直後に現れた (つまりファイルの最後の文字が'\n' だった) 場合は x 行目に現れたとみなす.

- `int char_stream_cur_column(char_stream_t` s`)`

  文字ストリーム s の現在の列番号を返す．現在の列番号とは，現在の文字が，現在の行の何番目の文字であるかを示す番号で，各行の最初の文字は 1 列目.

- char_stream_next_char(char_stream_t s)

  ファイルから 1 文字読み込んで，文字ストリーム s の現在の文字をそれにセットする．

この部品自身を作るためにファイルを読むライブラリとしては，fopen, fgetc を使えばよい．

この部品が完成すれば，目的のプログラムは以下のようにあっさりできるはずである．ほとんどの部分はエラーの検査で，本質的なコードは while ループ 3 行だけである．

行数を数えるプログラム

```
< char_stream を実装しているコード (省略) >

void line_count(char * filename)
{
  char_stream_t cs = mk_char_stream(filename); /* char_stream を作る */
  if (cs == 0) {
    fprintf(stderr, "could not open %s\n", filename);
    exit(1);
  }
  while (char_stream_cur_char(cs) != EOF) {  /* EOF が出るまで読んでいく */
    char_stream_next_char(cs);
  }
  /* char_stream の仕様により，EOF の行番号がファイルの行数 */
  printf("%d lines\n", char_stream_cur_line(cs));
}

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

'a' の位置を表示するプログラム

```
< char_stream を実装しているコード (省略) >

void find_a(char * filename)
{
  char_stream_t cs = mk_char_stream(filename); /* char_stream を作る */
  if (cs == 0) {
    fprintf(stderr, "could not open %s\n", filename);
    exit(1);
  }
  while (char_stream_cur_char(cs) != EOF) {  /* EOF が出るまで読んでいく */
```

```
    if (char_stream_cur_char(cs) == 'a') {
      int l = char_stream_cur_line(cs);
      int c = char_stream_cur_column(cs);
      printf("line %d column %d\n", l, c);
    }
    char_stream_next_char(cs);
  }
}

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

小さな例だが，この例で大事なポイントは，

- char_stream という部品と，それに付随する関数の動作が明確に定められている．
- それを使うコードは，各関数の中身はおろか，構造体の中身が何であるのかすら知らなくても，char_stream を使うことができる．
- 結果として，line_count や find_a というプログラム片は，それを見ただけでその意図するところや，それが正しいことが分かりやすくなっている．

もちろん char_stream の中にバグがあるかもしれないので，上のプログラムが全体として正しく動くかどうかは，上の断片を見ただけではわからないわけだが，仮にプログラムがおかしな動作をしたとしても，おそらく line_count 自身にバグはなく，あるとしたら char_stream の中にある，ということがわかる．そもそもその reasoning 自体が，char_stream がどう動作すべきかがきちんと明言されていないと不可能であることに注意してもらいたい．

この例を通して，そこそこの大きさのプログラムを組み立てていく際のやり方をもう少し明文化すると，

- プログラムを「意味のある動作をする」部品の集まりに分割する
- 部品とは，大雑把に言って，新しいデータ型 (多くの場合構造体) と，それに付随する関数である．C 言語なら typedef と関数の集まりとして構成されるが，Java, C++, Python などのオブジェクト指向言語では，class がまさにそれを提供する．
- 部品を構成する各関数の「意味」が明確に定義されていなくてはならず，かつ使うほうにとって使いやすい単純なものでなくてはならない．この際，そのデータ型の中身を詳細に知らなくても，その意味がきちんと通じるように書かれていなくてはならない (**抽象化**の原則)．

思えば，C や Java で提供されているライブラリも，そのように設計されている．各関数の動作をマニュアルで読めばその動作や，それをどう使えばよいかが分かるようになっている．決して，fread 関数の中身がどう実装されているかとか，FILE 構造体の中身にどんな要素があるかなどを知らなくても「使える」ようになっている．

自分がそこそこのサイズのプログラムを作る際も，万事「部品を作る．いくつか部品ができたら，その中身を忘れてそれを使う部品をまた作る」という具合に組み立てていくのである．決して，自分がプログラムのすべてを一時に把握できるとは思わないことである．現実的には，何を部品と部品の境界線と思うか，という概念自体がプログラム作成中に変化することもある．大事なことは，部品と部品の境界線がわかりにくくなってきたら，それを一種の危険信号と思ってうことである．そして再び部品の機能と意味を明確化する，という試行錯誤を恐れずに繰り返すということである．

5 分割コンパイル

以下では行を数えるプログラムを説明に用いる．

今，プログラムが完成して，1 つのファイル中に，char_stream の部品と，line_count, main が書かれているとする．プログラムがさらに大きくなってくると，ファイルを分割したくなってくる．この際には,「部品」ごとにファイルを分けるのが自然というものであろう．そこでプログラムを，char_stream を構成する部分と，それ以外の部分に分けてみる．

char_stream を実装するコードを cs.c というファイルに格納し，それを使うほうを lc.c としてみよう．単に，ファイルを文字通り途中で分割しただけだと lc.c はいろいろなエラーメッセージを出して失敗するはずである．たとえば gcc であれば，

```
prompt% gcc -c lc.c
lc.c: In function ‘line_count’:
lc.c:3: error: ‘char_stream_t’ undeclared (first use in this function)
lc.c:3: error: (Each undeclared identifier is reported only once
lc.c:3: error: for each function it appears in.)
lc.c:3: error: syntax error before "cs"
lc.c:4: error: ‘cs’ undeclared (first use in this function)
lc.c:5: error: ‘stderr’ undeclared (first use in this function)
lc.c:8: error: ‘EOF’ undeclared (first use in this function)
lc.c: In function ‘main’:
lc.c:17: error: ‘stderr’ undeclared (first use in this function)
```

のようなエラーが出るであろう．オプション-c は，プログラムのリンクはしない，というオプションである．意味が分からない人は分割コンパイルでおのおののファイルをコンパイルする際には指定する，と思って置けばよい．

最初に出たエラーメッセージ

```
lc.c:3: error: ‘char_stream_t’ undeclared (first use in this function)
```

は，char_stream_t などという名前 (型の名前) をコンパイラが知らない，と言っている．それもそのはずで，ファイルを分割して lc.c だけをコンパイラが見てコンパイルしているのだから，コンパイラがその定義がどこにあるかを知らなくても当然である．

しかし，だからといって typedef char_stream を lc.c に移すわけにも行かない．せっかくの部品の境界線を無視しているし，そもそもそうすると，cs.c の方で同じエラーを食らってしまうだけである．帰結は，typedef char_stream は両方になくてはならない，というものである．しかし，本当に両方にそのコードがあるのは望ましくない．データ構造を変更したら，両方を変更しなくてはならない．

そのために用いられるのが#include 節である．

```
#include <ファイル名>
```

と書くことで，ソースファイル中に別のソースファイルの中身を埋め込むことができる．ファイルの中身を文字列として，実際にそこに展開している，と思って置けばよい．よって今の場合，typedef は第 3 のファイルに書いておき，それを両者から#include すればよい．C の慣習では，cs.h というファイルを作って typedef 部分をそこに置く．

ここでコンパイルをしてもまだ以下のようなメッセージが出ることだろう．

```
prompt% gcc -c lc.c
lc.c: In function ‘line_count’:
lc.c:5: warning: initialization makes pointer from integer without a cast
```

また，gcc の警告レベルを上げる (いろいろな警告を出すようにするオプションで，実は常につけておくことを薦める) ともっといろいろと出るようになる．

```
prompt% gcc -Wall -c lc.c
lc.c: In function ‘line_count’:
lc.c:5: warning: implicit declaration of function ‘mk_char_stream’
lc.c:5: warning: initialization makes pointer from integer without a cast
lc.c:10: warning: implicit declaration of function ‘cur_char’
lc.c:11: warning: implicit declaration of function ‘next_char’
lc.c:13: warning: implicit declaration of function ‘cur_line’
```

lc.c の 5 行目は実はこのようになっている．

```
  char_stream_t cs = mk_char_stream(filename);
```

最初の警告

```
lc.c:5: warning: implicit declaration of function ‘mk_char_stream’
```

は，mk_char_stream というファイルが，定義される前に使われた，というメッセージである (正確には定義ではなくて宣言というのだが，違いは後ほど述べる)．それもそのはずで，ファイルを分割する以前は mk_char_stream がファイルの上の方で定義されたあと，ここで使われていた．コンパイラはファイルを前のほうから読んでいき，定義された関数に関してはその引数の数や型，返り値

の方を覚えておく．そして，呼び出された際に，渡されている引数の数と型が合致しているかどうかを検査するのである．ところが今はその定義が別のファイルに移動してしまったので，使われる前に定義が出てきていない，と言うわけである．

これは先ほどの typedef がないというエラーと同じ問題である．実は C の場合，これはエラーにはならず，warning (警告) になる．つまりこのままでも，プログラムを動かすことは可能である．しかしそうすると，引数の数や型を間違えたままプログラムを動かすことになる可能性があって，危険である．この警告はエラーと同値と受け取っておくのが良い (C++ではエラーになる)．

解決法としては，cs.c から，関数の定義を cs.h に移すことが考えられるがそれをすべての関数にやってしまっては元の木阿弥．何のことはない，lc.c から cs.c をまるごと#include しているのと同じことになってしまう．「それでもファイルがちゃんと分割できてるんだからいいんじゃないの?」と思うかもしれない．実を言うとこの例に関してはそうともいえるのだが，たとえば lc.c 以外にも cs.c の機能を使いたいファイル (x.c としよう) が現れたときに問題となる．もし両者が cs.c を #include すると，lc.c と，x.c の両方に関数の定義が現れることになってしまい，リンク時に重複定義と言うエラーになってしまう．

そういうわけで，ヘッダファイル (他から include されるファイル) 中に関数や変数の定義を書くことは通常行わない．そうする代わりに関数の引数や型だけを「宣言」する構文が用意されており，それをヘッダファイル中に書く．関数の宣言の構文は，関数の定義から，本体を取り除き，; で終わらせるものである．つまり，

```
char_stream_t mk_char_stream(char * filename);
```

のように書いておく．こうするとコンパイラは以下のように動く．

- この宣言以降，mk_char_stream が呼び出されているのを見ると，引数の数や型が検査され，間違っていればコンパイル時にエラーが出る．
- 同時に，関数呼び出しの結果の型が，宣言されている型 (今の場合 char_stream_t) とみなされ，それが間違った型の変数に代入されていないか，などの検査が行われる．
- mk_char_stream が後に定義されているのを見ると，宣言と定義で型が合致しているかどうかが検査され，間違っていればコンパイル時にエラーが出る．

ある部品のヘッダファイル中に関数の宣言があると，その部品を使うファイルは，そのヘッダファイルを#include すれば，宣言と呼び出しで型が食い違う心配がなくなる．一方その部品の中身を記述するファイルでも#include すれば，宣言と定義とで型が食い違う心配もなくなる．結果として，呼び出しと定義とで型が食い違う心配がなくなるのである．つまり，型の間違った関数呼び出しをする心配がなくなる．

これは，ほとんどの言語で当然のように保障されていることである．Python はそうではないが，引数の数などは実行時に検査されて，間違っていればその旨のエラーが表示される．C は，宣言をきちんと書かなかった場合 (つまり，上で見たような警告を放置して実行した場合)，関数呼び出し時に渡される引数の数や型が，関数の定義と一致しているという保証はなくなり，しかも実行時にそのエラーが検出されることもなく実行が続けられるという世にも恐ろしい言語である．たとえば引数を 3 つ受け取る関数に引数を 2 つしか渡さなければ，3 つ目の引数にはめちゃくちゃな引数が渡される．

まとめると，分割コンパイルのためのファイルのおおまかな分割は以下のようになる．

cs.h 文字ストリームの型定義`typedef struct char_stream { ... } ...`, および`mk_char_stream`, `char_stream_next_char`, `char_stream_cur_line` の関数の宣言．

cs.c `cs.h` を`#include` した上で，上記の各関数を定義．

lc.c `cs.h` を`#include` した上で，`line_count` および `main` を定義．

6 makeによるコンパイル作業の簡略化

6.1 動機付け

さて，やっと正しくファイルが分割されたところで，実際に実行可能ファイルを生成 (しばしばプログラムの構築，ビルド (build) などと呼ばれる) してみよう．

まず，単純には以下のようなコマンドでそれが可能である．

```
gcc -o lc cs.c lc.c
```

マイクロソフトならば`gcc` のかわりに `cl` を使う．`-o` は，出力ファイル名を指定するオプションで，上記はそれを `lc` としている．ただし，この方法はプログラムを少し修正するたびにすべてのファイルをコンパイルしなおす羽目になり，大きなプログラムでは時間の無駄になる (本演習で作るプログラムでは，さして気になる時間ではないが)．

これを防ぐためには各ファイルを別途コンパイルしておく方法がある．そのためのビルド手順は以下のようになる．

1. `gcc -c cs.c` `# cs.c のコンパイル．cs.o が生成される`
2. `gcc -c lc.c` `# lc.c のコンパイル．lc.o が生成される`
3. `gcc -o lc lc.o cs.o` `# リンク．lc が生成される`

5節でも一度述べたとおり，`-c` は「リンクをしない」というオプションである．これは,「オブジェクトファイル」という，`.c` 内の関数に対する機械語命令列が格納したファイルを生成する．オブジェクトファイルそれ自体は実行可能なプログラムではない (`main` を含んでいない場合もあるので当然である)．

最後の行で実行可能ファイルを生成しているが，`.c` ではなく，その前の 2 行で作った`.o` ファイルを引数に指定している．ここで行われるのはプログラムのコンパイル (機械語への翻訳) ではなく，オブジェクトファイルをひとつにまとめて実行可能ファイルを作る,「リンク」と呼ばれる作業である．

一般にファイルを分割した場合，プログラムを生成する手順は,

- 各`.c` ファイルを`-c` オプションをつけながらコンパイルし，オブジェクトファイルを生成する．
- 生成された`.o` ファイルをリンクする．

となる．

そして，ある.c ファイルを変更したらその.c ファイルのコンパイルと，リンクだけをやり直せばよい．言葉で書くと確かによく見えるが，コンパイルのたびに，実際に，上のコマンドを手で打ち込むのはかえって面倒が大きい．しかも，どのしばらくファイルをいじった後で，どのファイルが変更されたかなどを覚えておくのはさらに面倒である．

make はこのようなプログラム構築の煩雑さを解消してくれるツールである．実は，Microsoft を含め「統合開発環境」と呼ばれているツールにはこの機能が組み込まれていることがほとんどで，「ビルド」というコマンドを一発発行するだけで，必要なファイルのコンパイルとリンクを行ってくれる．その意味では make は原始的なコマンドラインコンパイラの不便さを補うだけのものともいえるのだが，一方で make は (cygwin を含む) UNIX 環境で広く使われており，ほとんどのソースで配布されているプログラムも，make を使って構築するようにパッケージが作られている．実際，Microsoft の環境もコマンドラインコンパイラや独自の make ツールも提供している．したがって，開発環境に依存しないツールとして習得しておくことには価値があるだろう．本演習も，やがて 5 つや 6 つくらいのファイルに分かれたプロジェクトになっていくので，make を用いてコンパイルするのは大いに意味がある．

6.2 make について

さていよいよ make の説明に入る．make は，`Makefile` と言う名前のファイルに「プログラムを生成する規則 (プログラムの名前やコマンドライン)」を記述しておくと，それを読み取って自動的にプログラムのビルドを行ってくれるツールである．先に結論を見せると，われわれのプログラムのビルドを行う最も単純な `Makefile` は以下の内容である．

```
lc : lc.o cs.o
gcc -o lc lc.o cs.o
```

2 行目のインデントはタブである．スペースではだめなので注意．

このファイルをソースファイルと同じディレクトリに置き，`lc.o`, `cs.o` が存在しない状態で，

```
prompt% make
```

とコマンドを実行すれば，以下のようにプログラムがビルドされる．

```
prompt% make
gcc    -c -o lc.o lc.c
gcc    -c -o cs.o cs.c
gcc -o lc lc.o cs.o
```

ここで，`cs.o` だけを消去してもう一度 make をしてみよう．

```
prompt% rm cs.o
prompt% make
gcc    -c -o cs.o cs.c
gcc -o lc lc.o cs.o
```

面白いことに，lc.c のコンパイルは行われずに，cs.c のコンパイルだけが行われる．さらに，cs.o を消去しなくても，cs.c を変更すると，同様に cs.c の再コンパイルが行われる．

```
prompt% <エディタで cs.c を修正>
prompt% make
gcc    -c -o cs.o cs.c
gcc -o lc lc.o cs.o
```

この状態でもう一度 make を起動すれば,

```
prompt% make
make: ‘lc’ is up to date.
```

すなわち，lc はすでにビルドされており，その後ソースファイルの更新なども行われていないので，「もう何もすることはない」と言っている．このように make は，実行可能ファイルをビルドするだけでなく，そのために必要最低限のコマンドを起動する，という仕組みを持っているのである．

Makefile をどう記述したらよいかを述べるため，もう一度 Makefile の中身に戻ってみる．

```
lc : lc.o cs.o
gcc -o lc lc.o cs.o
```

インデントの入っていない行 (1 行目) は,

目標ファイル : 依存ファイルのリスト

という文法をしている．目標ファイルとは，まさに作るべきファイルであり，依存ファイルとは，その目標ファイルが依存している (大雑把に言えば，その目標ファイルを作るためには，依存ファイルが必要である) ということを示している．この例では，実行可能ファイル lc を作るには，lc.o, cs.o が必要であるということを書いている．一方その後の，インデントで始まる行は，目標ファイルを作るにはどういうコマンドを実行したらよいかが書いてある．この例では，まさしく

```
gcc -o lc lc.o cs.o
```

を実行せよと書かれている．上記で make をした際の，最後の行はこの記述によって実行されている．

少し不思議なのは，なぜそれ以前の，.c から .o を作るコマンド

```
gcc    -c -o lc.o lc.c
gcc    -c -o cs.o cs.c
```

は，Makefile に何もかかれていないのにもかかわらず実行されたのか，であろう．

その理由は，make はプログラムを構築するためのツールとして，そのようなルールを組み込みで持っている，ということである．具体的には，上で見せたとおり，.c から .o を生成するのに，C コンパイラを-c オプションつきで起動すればよいということを知っているのである．他にも .cc (C++) から .o を生成するルールなども知っている．

6.3 その他の make の有用な機能

変数 上の Makefile を実際に書いてみると，`lc.o cs.o` を 2 度書いている．このような，何度も同じことを書くのを避けるために，変数を定義して使うことができる．具体的には上記の例を以下のように書き換えると良い．この例ではありがたみを感じないかもしれないが，ファイルの数が 3, 4, ··· と増えるにつれて重宝するようになるであろう．

```
OBJS = lc.o cs.o
lc : $(OBJS)
gcc -o lc $(OBJS)
```

複数の目標とその指定 Makefile 内に複数の目標ファイルと，それを構築するためのルールを書くことができる．make のコマンドラインで，明示的に構築したい目標を指定することができる．たとえば今のわれわれの Makefile でも，

```
make cs.o
```

のように make を起動すれば，`cs.c` のコンパイルのみが行われる．make を無引数で指定した場合，Makefile 内の最初の目標ファイルを構築するべく，コマンドが起動される．

われわれの当面の目的では，Makefile はただひとつの目標を構築することができれば十分だが，それでも以下のような目標を，Makefile の終わりのほうに記述しておくと有用である．

```
clean :
rm -f $(OBJS)
```

これは，先ほどの OBJS 変数の定義と合わせて，すべてのオブジェクトファイルを削除するというもので，これを記述した上で，

```
make clean
```

を行うとオブジェクトファイルが削除される．これはあらためてすべてのファイルをコンパイルしなおしたいときなどに有用である．

注意としては，`rm -f $(OBJS)` を実行したところで，実際に clean というファイルが作られるわけではない．make は単に，clean というファイルがないので，指示されたとおり `rm -f $(OBJS)` というコマンドを実行しているに過ぎない．

コンパイル時のオプション `.c` から `.o` を生成する際の方法は make が知っていると述べたが，その際に自分でオプションを指定したくなることがある．そのために make は組み込みの変数 CFLAGS を用意しており，ここに指定された文字列は C コンパイラを起動する際のオプションとして使われる．gcc を使うのであれば，`-Wall` という，警告レベルをあげるオプションをすすめておく．

```
CFLAGS = -Wall
OBJS = lc.o cs.o
lc : $(OBJS)
gcc -o lc $(OBJS)
```

このようにMakefileを書いておけばコンパイルは以下のように行われる．

```
prompt% make
gcc -Wall   -c -o lc.o lc.c
gcc -Wall   -c -o cs.o cs.c
gcc -o lc lc.o cs.o
```

C++コンパイラのオプションには`CXXFLAGS`を用いる．

7 Emacsを統合環境として使う

Emacsはプログラムを編集するのみならず，いろいろなコマンドによって，プログラム開発のための統合環境のように使うことが可能である．プログラムを開発する際に重要なものは以下である．

7.1 M-x compile

Emacs内でプログラムをコンパイルするコマンドを実行し，エラーメッセージが現れたファイルと行へ，カーソルを飛ばすことができる．Emacsについてはいろいろな書籍が出ているので適当な入門書を読むことをお勧めする．また，Emacsがインストールされていればおそらく，M-x info コマンドによってマニュアルを参照できるだろう．

さて，M-x compileは以下のように使う．

1. プログラムを編集しているバッファにおいて，M-x compileコマンドを実行すると，ミニバッファに

   ```
   Compile command: make -k
   ```

   という文字列が現れ，入力を促される．これは，'make -k' というコマンドを実行しようとしており，自由に編集してよい．節で述べたようにMakefileが書いてあり，makeコマンドでプログラムをコンパイルする体制が整っていればそのままリターンを押せばよい．まだの場合，ここに，

   ```
   gcc lc.c
   ```

   などのコマンドを直接書いても良い．

   いずれにせよ，`Compile command:` 以降に書いてあるコマンドが実行される．

2. ここまでは，普通に別のウィンドウでコンパイルコマンドを実行した場合と大差がなく，あまりありがたみを感じないかもしれないが，コンパイルエラーが出た後が大違いになる．具体的には，

   ```
   C-x `
   ```

というキー (C-x は，Control キーを押しながら x を押すことを示す．つまり全体として Control キーを押しながら x を押し，その後単独で ‘ (バッククオート) を押す) によって，エラーメッセージが現れたファイルと行に飛ぶことができる．その後も同じキー操作を繰り返すことで次のエラーメッセージに順次飛んでいくことができる．

これは，コンパイラのエラーメッセージからファイルと行番号を読み取ってそこへ自分で飛んでいくのとは大違いである．

7.2 M-x shell

コンパイルのために Emacs を離れずにすむようになったので，できたプログラムの実行も Emacs の中で行えるようにしたくなる．それには，M-x shell というコマンドを使う．使い方は簡単で，M-x shell というコマンドを実行するとシェルが Emacs の新しいバッファ内で起動される．これは通常のシェルとまったく同様だが，おまけの機能として以下のような機能がある．

ヒストリー機能: プロンプトにポイントを置いた状態で M-p によって，一つ前のコマンドラインがプロンプト以降に挿入される．

再実行機能: シェルは Emacs のバッファ内で走るので，通常のバッファ同様，カーソルの移動ができる．過去の，コマンドを入力した行へカーソルを移動して，リターンキーを押すと，そのコマンドがもう一度実行される．

7.3 M-x gdb

デバッグも同様である．デバッグについてはいずれまた説明するが，M-x gdb とすると，デバッガを Emacs のシェル内で走らせることができ，かつ，ブレークポイントなどでプログラムをとめた際，対応するソースファイルと行へ，自動的にカーソルを飛ばすことができる．使い方は以下のとおり．

1. プログラムをデバッグシンボルつきでコンパイルする．gcc では `-g` オプションを指定してコンパイルすると，プログラムがデバッグシンボルつきでコンパイルされる．make を使っているのであれば，`CFLAGS` に，`-g` を追加する．

2. ソースファイルのあるバッファで M-x gdb コマンドを実行すると，ミニバッファに

   ```
   Run gdb (like this): gdb
   ```

   という行が現れる．`gdb` の後に，実行可能ファイル名を追加して，

   ```
   Run gdb (like this): gdb lc
   ```

   のようにしてリターンキーを押すと，新しいバッファが現れてそこで gdb が実行される．

3. 以降は gdb の使い方になるので詳しいことはまたの機会に説明する．よく知っている人は，`main` にブレークポイントを置いて (`b main`)，プログラムを実行 (`r`) させてみるとよい．`main` の先頭でプログラムが停止し，かつそこへ自動的にカーソルが飛ぶのが観察できるだろう．

注: Windows 環境 (Meadow + cygwin の組み合わせ) でこのカーソルジャンプが機能しない問題がある．cygwin-mount.el というプログラムを導入すると解決する．演習 HP 参照.

8 CVS

演習 HP を参照しながら以下をおこなう．目標はチーム間で簡単にファイルの更新をやり取りできるようになることである．ファイル数が 3 つほどにもなれば，ありがたみがわかるだろう.

以下で CVS サーバとは，marten.logos.ic.i.u-tokyo.ac.jp のことである.

- (CVS サーバ上) 自分のレポジトリを作成.

  ```
  cvs -d /home/ユーザ名/CVS init
  ```

- (自分のマシン上) 適当なモジュールを登録 (import).

  適当なディレクトリ (= d とする) に数個のファイルを入れておく.

  ```
  export CVS_RSH=ssh
  export CVSROOT=:ext:ユーザ名@marten.logos.ic.i.u-tokyo.ac.jp:/home/ユーザ名/CVS
  cd d
  cvs import -m "my first cvs module"  モジュール名 V R
  ```

  最初の 2 行は.bashrc などに書いておくとよい.

  V, R は適当な文字列で，何でもよいのだが，V (vendor-tag という) にはユーザ名，R (release-tag という) には init などの文字列を入れておくのが慣習.

  これで，モジュールのファイルがサーバ (レポジトリという) 上に登録された．CVS サーバ上で，/home/ユーザ名/CVS ディレクトリの下を眺めて見るとよい.

- (**重要:** 自分のマシン上) あらためてそのモジュールをチェックアウト.

  ```
  export CVS_RSH=ssh
  export CVSROOT=:ext:ユーザ名@marten.logos.ic.i.u-tokyo.ac.jp:/home/ユーザ名/CVS
  cd 今後作業をする適当なディレクトリ
  cvs co モジュール名
  ```

  とすると，モジュールのディレクトリまるごとのコピーがサーバからダウンロードされるはずである.

  以降は，**先ほど登録時に用いたディレクトリ** (d) **はなく，co によってダウンロードされたディレクトリの方で**作業を行う．登録時に用いた方は消してしまってもよい.

  チームメイトがいる場合，その人も co することで，同一のコピーを二人が手にすることができる.

- (自分のマシン上) 更新とその反映

  適当にファイルを更新してみる.

  ファイルを更新したら,

  ```
  cvs commit -m "コメント"
  ```

  とすると，変更内容がサーバ上に書き込まれる.

  ほかの人が commit した変更内容を自分のコピーに反映するには,

  ```
  cvs update -d
  ```

  いったんプロジェクトが始まったら，通常はこの二つをコマンドだけを使う.

  二人が同じファイルを更新した場合，どのようなことがおこるか? ⇒ 口頭説明とこれからの経験で学ぶ.

- (自分のマシン上) ファイルの追加

  作業ディレクトリにファイルを新たに作って，その後で `cvs commit` しても，自動的にそのファイルがレポジトリに追加されるわけではない.

  ```
  cvs add ファイル名
  ```

  とした後で，あらためて `cvs commit` を行う．ディレクトリの追加も同様だが，この場合は改めての `cvs commit` は必要ない.

以上で cvs の基本操作を理解したら，このたび作ったモジュールを，Makefile などとともに一式, `char_stream` という名前で登録すること (本日の課題はこれによって提出とみなす).