B演習 (言語処理系演習) 資料 (4)
構文解析器

田浦

2006/11/6

1 概要

字句解析器によって，文字列であったプログラムが，字句の列として認識される．構文解析器はその字句の列を受け取ってそれが構文的に正しい字句の並びであるかどうかを判定し，正しくなければエラーを表示する，正しければ，読み込まれたプログラムを表現するデータ構造 (構文木) を作成する．

2 構文の定義

2.1 概要

構文の定義の詳細は演習の HP に掲載しているので適宜そちらを参照すること．ここではその読み方を例を挙げながら説明する．以下は while 文の文法を定義している部分の抜粋である．

```
while_stmt ::= "while" expression ":" suite
```

この式が意味するところをインフォーマルに述べると，

> while_stmt とは，"while", expression, ":", suite を並べたもの

ということである．ここで，"while", ":"というのは，文字通り while, :などの，ひとつの字句をあらわす (すなわち，字句解析のところで定義した TOK_WHILE や，TOK_COLON などに相当する)．

一方，expression や suite は，別途，やはり上のような形式で定義されている字句の列であって，たとえば suite は以下のように定義されている．

```
suite ::= NEWLINE INDENT statement+ DEDENT
```

すなわち suite とは，

> 改行，字下げ，文 (statement) のひとつ以上の並び，逆字下げ

を並べたものと言うことである．ここでも statement は，やはり別の場所で定義されている．

statement や，statement の定義中に出てくる別の記号など，すべての記号の定義は文法定義のどこかに書かれており (書かれていなければ文法定義書の間違いですのでご報告を)，全体として，このようなたくさんの定義式をつなぎ合わせて，最終的には，

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

が,「正しいプログラム」とみなされるべき文字列を定義している．縦棒 (|) は,「その前後のどちらか」をあらわしており，この定義は,

> `file_input` とは,「改行または文 (statement) のどちらか」が 0 個以上並び，それが終わったら入力全体の終わりとなるものである

と言う意味である．

2.2 文脈自由文法

ここで使われているような文法の定義を見せられれば，いくつかの記法上の慣習さえ知っていればほとんど説明されなくてもその意味するところが分かるであろう．実際プログラミング言語のみならず，ソフトウェアの設定ファイルの書き方や，インターネットを流れるデータのフォーマットなど，ありとあらゆる「フォーマット」がこのような記法で定義されている．

それはフォーマルには「文脈自由文法」と呼ばれる．ソフトウェア基礎論の授業などで出てくるはずである．ここではインフォーマルな解説をする．文脈自由文法とは,

$$A \to B_1 \cdots B_n$$

という形の規則 (生成規則) を並べて，目的の文字列 (ここでは字句の列) の集合を定義する枠組みである．ここで，いくつかの注意としては,

- A は記号である．
- B_i は，どこかの生成規則の左辺に現れている記号 (非終端記号と言う) か，字句名 (終端記号という) のどちらかである．
- 同じ記号に対する生成規則は複数あってもかまわない

我々の文法規則の記法では上を

$$A \texttt{ ::= } B_1 \cdots B_n$$

と書いていたのである．また，我々の文法規則には，0 回以上の繰り返しを表す*などのいくつかの略記法が導入されているが，それらは容易に正式な表記に書き換えられるもので，本質になんら影響はない (後述)．

生成規則

$$A \to B_1 \cdots B_n$$

の意味をもう少し正式に考えてみるとそれは以下のようなことである.

> B_1 とみなされるある文字列 (s_1) と,
> B_2 とみなされるある文字列 (s_2) と,
> $\cdots$
> B_n とみなされるある文字列 (s_n)
> があったならば, それらを連結した文字列 $s = s_1 \cdots s_n$
> は, A とみなす.

これは後に, 文法規則から構文解析器を導く際の主要な考え方になる.

2.3 mini-Python の文法定義の表記と正式な文脈自由文法の表記の微妙な際について

mini-Python の文法記述 (のみならず, 世の中で使われる多くの文法記述) には, 正式な文脈自由文法の記述といくつか違いがある. それは表面的なものに過ぎないので, ここで説明しておく.

選択肢を示す縦棒 (|): mini-Python では文法規則の右辺に, いくつかの選択肢を縦棒 (|) で区切りながら示すことがある. たとえば,

```
compound_stmt ::= if_stmt
                | while_stmt
                | for_stmt
                | funcdef
```

のように. しかしこれを, 正式な文脈自由文法の記法に直すのは簡単で,

```
compound_stmt → if_stmt
compound_stmt → while_stmt
compound_stmt → for_stmt
compound_stmt → funcdef
```

でよい.

省略可を示す大括弧 ([...]): mini-Python の文法にはしばしば, 右辺の一部を [$\cdots$] で囲んだもの現れる. たとえば,

```
return_stmt ::= "return" [expression]
```

など. この [$\cdots$] は, $\cdots$ の部分が省略可である, つまりなくてもよいということである. これは, 以下と同じことである.

```
return_stmt ::= "return" (expression | ϵ)
```

ただし，ϵ は「空文字列」をあらわす特別な記号とする．これは最終的に正式な文脈自由文法で書けば，

```
return_stmt → "return" return_rest
return_rest → ϵ
return_rest → expression
```

ということである．

繰り返しを示す*，+: mini-Python では，右辺の一部の記号 (ないし複数の記号を括弧でくくったもの) の後ろに*や+をつけることがある．たとえば，

```
parameter_list ::= identifier ("," identifier)*
```

のように．*, +はそれぞれ，直前の記号 (ないし複数の記号を括弧でくくったもの) の 0 回以上，または 1 回以上の繰り返しを表している．つまりこの例は，parameter_list が，identifier で始まり，その後"," identifier が 0 回以上繰り返されたものであることを示している．この例は以下の文脈自由文法の記述に置き換えられる．

```
parameter_list → identifier parameter_list_rest
parameter_list_rest → ϵ
parameter_list_rest → "," identifier parameter_list_rest
```

実際にこの書き換え作業を行えという意味ではない．ここでのポイントは，結局我々の問題が，文脈自由文法で書かれた文法の構文解析という問題に帰着された，ということである．

3 構文木: プログラムを表すデータ構造

構文木はプログラムを表すデータ構造で，つまり，そのデータ構造だけで元のプログラムが (空白の数などの，実行に決して影響を与えない部分の違いを除き) そっくり再現できるようなデータ構造である．もちろん，元のプログラムが再現できるだけならば，元のプログラムの文字列をそのまま構文木と称してもよさそうなものだが，これでは実行のために不便すぎる．構文木は同時に，プログラムの構文的な「構造」を簡便にあらわしたものでなくてはならない．

たとえば，「文」をあらわす構文木は，それがどの種類の文なのか (while 文，for 文，etc.) を簡単に区別できなくてはいけないし，while 文であればさらに，条件部にどんな式が来ているのか，本体にはどんな文 (の並び) が来ているのか，などがたちどころに分かる構造でなくてはならない．

そのためにたくさんの C の構造体を typedef する．一つ一つの定義は難しいことは何もなく，その文なり式なりを構成する要素をきちんと構造体の要素としてもたせてやればよい．たとえば「while 文」の定義は以下のようになる．

```
typedef struct stmt_while
{ /* while e: b */
```

```
  expr_t e;
  stmt_vec_t b;
} stmt_while, * stmt_while_t;
```

while 文の文法：

```
while_stmt ::=
             "while" expression ":" suite
```

を見ても分かるように，while 文は，条件式 (expression) と，本体 (suite) から成る．それに対応して，e と b という要素を作っているだけである．次は for 文の例：

```
typedef struct stmt_for
{ /* for x in e: s */
  char * x;
  expr_t e;
  stmt_vec_t b; /* body */
} stmt_for, * stmt_for_t;
```

これも，以下の for 文の定義から自然に導かれている．

```
for_stmt ::= "for" target "in" expression ":" suite
```

文全体は様々な場合の union である．

```
typedef struct stmt
{
  stmt_kind_t kind;
  src_pos_t pos;
  union {
    expr_t e; /* expression, return, print statement, del */
    stmt_assignment a;
    if_branch_vec_t i;
    stmt_while w;
    stmt_for f;
    stmt_fundef d;
    char * g; /* global x */
  } u;
} stmt;
```

typedef だけでなく，各構文ごとにそれを作成する関数を作る．たとえば while 文に対しては，

```
stmt_t mk_stmt_for(char * x, expr_t e, stmt_vec_t b, src_pos_t pos)
{
  stmt_t s = stmt_alloc(stmt_kind_for, pos);
```

```
  s->u.f.x = x;
  s->u.f.e = e;
  s->u.f.b = b;
  return s;
}
```

のような具合である．ここで stmt_alloc は，

```
stmt_t stmt_alloc(stmt_kind_t k, src_pos_t pos)
{
  stmt_t s = (stmt_t)my_malloc(sizeof(stmt));
  s->kind = k;
  s->pos = pos;
  return s;
}
```

構文木の定義 (syntree.h, syntree.c) は演習 HP から提供する．

4 構文解析器のインタフェース (外部インタフェース)

構文解析器の外部インタフェースと呼ぶべきものは，以下のみである．

- file_input_t parse_file_input(tokenizer_t t)

これは字句解析器 t から字句を (この関数が呼ばれた時点での t の tok を最初の字句として) 次々と読み込み，途中で構文エラーが見つかればプログラムを終了し，そうでなければプログラム全体をあらわすデータ構造 file_input_t (syntree.h で定義) を作って返す．

5 構文解析器の中身の概要

5.1 基本枠組み

parse_file_input が構文解析への唯一の「外部からの」エントリポイントとなるわけだが，構文解析器の中身をのぞいてみると，それはたくさんの，parse_XXX という関数からなっている．

基本的には，A という非終端記号に対して，parse_A(tokenizer_t t) という関数があり，その動作は一様に以下のとおり．

- 字句解析器 t から字句を (この関数が呼ばれた時点での t の現在の字句を最初の字句として) 読み出して行き，A と見なせる字句列を読み込む．
- parse_A がリターンするとき，t の現在の字句は，A の一部として読み込まれた最後の字句の次の字句を保持している (分かりにくい日本語だが，言い換えると A の一部と見なされなかった最初の字句を保持している)．

- そして，A に応じた適切なデータ構造を生成して返す．そのために多くの場合，syntree.c に定義してある mk_XXX 関数のどれかを使う．
- もし，A と見なせる字句列が読み出されなければ，構文エラーを表示してプログラムを終了させる．

このような多数の parse_XYZ 関数たちから，構文解析器全体は作られている．

5.2 各 parse 関数の中身

さて，parse_A の中身はどう書いたらよいのか? それは A の生成規則から決まる．例によってまた while 文を例にとって説明しよう．

```
while_stmt ::=
             "while" expression ":" suite
suite ::=
             NEWLINE INDENT statement+ DEDENT
```

二つをまとめて書けば,

```
while_stmt ::=
             "while" expression ":" NEWLINE INDENT statement+ DEDENT
```

これに対応して，parse_stmt_while という関数がある．これが行うことは,

1. 現在の字句から始まり，"while" expression ":" NEWLINE INDENT statement+ DEDENT に合致する字句列までを読み込んでリターンする．いいかえると，リターン時には，現在の字句は この while 文の一部と見なされなかった最初の字句が保持されている
2. 無事 while 文が読み込めたら while 文をあらわす構文木を作って返す (mk_stmt_while を使う)
3. そうでなければ構文エラーを表示してプログラムを終了 (exit) する

これは以下のようにすっきりと書ける．

```
stmt_t parse_stmt_while(tokenizer_t t)
{
  int _ = eat(t, TOK_KW_WHILE);  /* 現在の字句が ‘‘while’’? OK なら次の字句へ進む. */
  expr_t e = parse_expression(t);/* 式を 1 個読み込む */
  stmt_vec_t b;
  eat(t, TOK_COLON);             /* 現在の字句が ‘‘:’’? OK なら次の字句へ進む. */
  eat(t, TOK_NEWLINE);  /* 現在の字句が NEWLINE? OK なら次の字句へ進む. */
  eat(t, TOK_INDENT);            /* 現在の字句が INDENT? OK なら次の字句へ進む. */
  b = parse_statement_list(t);   /* 文をいくつか読み込む */
  eat(t, TOK_DEDENT);            /* 現在の字句が DEDENT? OK なら次の字句へ進む. */
  return mk_stmt_while(e, b, e->pos); /* おめでとう */
}
```

この例でなんとなく,「A の生成規則から，比較的機械的な手段で構文解析器 (parse_A) が作れそうだ」と思ってもらえただろうか? 事実そのとおりで，大体において，A の生成規則が,

$$A \to B_1 B_2 \cdots B_n$$

であれば，おおまかには,

```
parse_A(tokenizer_t t)
{
  parse_B_1(t);
  parse_B_2(t);
   ...
  parse_B_n(t);
}
```

とやればよいのである．ただし，B_i が終端記号 (単なる字句) のときは，parse_B_i() は上記の eat(B_i) に置き換えられ，eat(X) の動作は,

- 現在の字句が X でなければ構文エラーを表示してプログラムを終了する．
- X ならば，字句解析器にたいして次の字句を取り出すよう指示する (next(t) を呼ぶ).

5.3 複数選択肢がある場合

重要な付け加えるべき点として，A の生成規則に複数の選択肢 (縦棒) が含まれている場合がある．たとえば,

```
compound_stmt ::=
            if_stmt
          | while_stmt
          | for_stmt
          | funcdef
```

のように．この場合の基本的な対処の仕方は,

現在の字句を見ることで，どの選択肢を選択したらよいのかを決める

ということである．つまり，この例であれば，tok_cur_token_kind(t) が，TOK_KW_IF であれば if_stmt を選ぶ，という具合である．したがって，上記の compound_stmt を認識する構文解析器の断片は以下のような感じになるだろう．

```
  switch (tok_cur_token_kind(k)) {
  case TOK_KW_IF:
    e = parse_stmt_if(t); break;
  case TOK_KW_WHILE:
```

```
    e = parse_stmt_while(t); break;
  case TOK_KW_FOR:
    e = parse_stmt_for(t); break;
  case TOK_KW_DEF:
    e = parse_stmt_fundef(t); break;
  default:
    parse_err(t);
  }
}
```

一般に選択肢があるところでは,「各選択肢の最初の字句になりうる字句の集合」を求め，現在の字句がどの集合に属しているかによって適切な選択肢を選ぶ.

ここで頭が働いている人であれば以下のような疑問を持つことだろう.

もしここで「可能性のある選択肢」が複数存在したらどうするのか?

言い換えると，二つの選択肢に対して，それらの最初の字句になりうる字句の集合に重なりが合ったらどうするのか，ということである.

先に結論を述べてしまうと，実は我々の文法ではそのようなことはない．そのように mini-Python (Python というべきか) の文法が設計されているのである．もちろん一般の文法ではそうとは限らない．一般の文脈自由文法の構文解析アルゴリズムとしては，夏学期に CKY アルゴリズムを習ったはずである．これは文の長さ N に対して N^3 の時間を必要とするアルゴリズムで，プログラミング言語の構文解析器としてはまず用いられない．ここで紹介するやり方では解析できない文法で，プログラミング言語の文法として現れる文法の構文解析については，コンパイラの教科書を参照してもらいたい.

5.4 左再帰による無限再帰呼び出しの注意

次の注意．ここで述べたような方式が通用しなくなるもうひとつの原因がある．それは生成規則の中に,

$$A \to A \cdots$$

のように，左辺と，右辺の先頭に同じ記号が現れている場合である．これは，ここで述べた方式で構文解析器を作っても無限ループ (正確には無限の再帰呼び出し) に陥ってしまう．このような文法を「左再帰」といって，同じ文法でも書き方によって左再帰になったりなくなったりするので注意が必要である.

実はこのようなあからさまな例だけでなく,

$$A \to B \cdots$$
$$B \to C \cdots | A \cdots$$

のようなものも，$A \to B \to A$ という無限の再帰呼び出しを起こし得るという点で，同様である．つまり，A を認識する関数 (parse_A) が，字句をひとつも読み込まないまま，再び parse_A を呼び出してはならないということである．

再帰がすべて問題なのではない．以下のような例は再帰を含んでいても，OK であることに注意．

$$A \to B + A$$

それは，parse_A が再帰的に parse_A を呼び出すまでには，最低でも必ずひとつの字句 (+) を読み込んでいるからである．こうなっている限り，再帰呼び出しをするごとに最低でもひとつの字句が読み込まれていることになるから，無限の再帰呼び出しに陥ることはないのである．

ここで述べた方式で構文解析ができる文法のことを LL(1) 文法と呼ぶ．

構文解析木に関してもヘッダ (parser.h) と, ごく一部分の記号に対する構文解析処理の本体を記したファイル (almost_empty_parser.c) を提供する

6 おまけ

構文解析器はここで述べたような枠組みで「手動で」作成する方法と，文脈自由文法の記述から構文解析器を自動生成するツールを用いて作成する方法がある．構文解析器を生成するツールは「コンパイラコンパイラ」などと呼ばれることもあり，代表的なものに yacc/bison がある (Unix に標準でインストールされている)．興味のある人は試してみると良い．

7 今週の課題

なるべくたくさんの Python プログラムに対し構文解析を行い，作られた構文木をただちに印字するプログラムを作り，構文解析器をテストする．構文木の印字には syntree.c にある print_file_input を用いればよい．mini-Python プログラムを入力し，結果として「同じプログラム」が出てくればよいわけである．入力と出力は同じプログラムではあるが，文字列としては異なる．たとえばコメントなどは削除され，空白なども正規化されることになる．実際に同じプログラムかどうかを機械的に，あるいは人間の目で判定するのは難しいので，ここで表示されたプログラムを再び構文解析にかけて同じことをする．今度は，入力と出力は文字列としてまったく同じものになるはずである．

つまり，以下のようなテストプログラムを書き，

```
int parser_test(char * filename)
{
  tokenizer_t t = mk_tokenizer(filename);
  file_input_t u = parse_file_input(t);
  print_file_input(u, stdout);
  return 0;
}

int main(int argc, char ** argv)
```

```
{
  parser_test(argv[1]);
  exit(0);
}
```

コンパイルして得られた構文解析プログラムを parser とでもして,

```
% ./parse a.py > b.py
% ./parse b.py > c.py
% diff b.py c.py
```

それぞれのファイルを目視して大体正しいことを確認した後，最後の diff コマンドが無言で終了すれ (つまり b.py の中身と c.py の中身が文字列として同一なら) ば OK である．

入力として与えるプログラム (上記の a.py) は，いくつかは簡単なものを自分で作ってみよ．また，正しいプログラム以外に構文エラーを含むプログラムもいくつも試してみよ．ここで構文エラーを含むプログラムを，確実にはじけるようにしておくことは，後々の開発効率を上げるためにも重要である．

テストプログラムを HP 上で公開する．

8 余談: コンテナ

8.1 C 言語における汎用コンテナの実装方法

これは余談とあるが，一般的に言って，有用，かつ実際に演習のコードを書く際に重宝する知識である．また，こちらで提供しているコード (syntree.c) 中でも使われている (expr_vec_t, stmt_vec_t などがそれに相当する) ので，解説しておいた方が良いと思い，解説する．

「コンテナ」とは日本語で「入れ物」という意味で，配列，リスト，ハッシュ表，スタックのように，中にいろいろなデータを格納するデータ構造のことである．それらのデータを C 言語で定義するには，通常構造体を使う．たとえば以下のようなインタフェースを持つ「整数のリスト」を作るとしよう．

- `typedef struct int_list { ... } * int_list_t`
- `int_list_t mk_int_list()`：空の整数リストを作る
- `void int_list_append(int_list_t` l, int x`)`：リスト l の末尾に x を加える
- `int int_list_get(int_list_t` l, int i`)`：リスト l の i 番目の要素を返す

この中をどう作るかは今は本題ではないので問わない．

さて問題は，次にたとえば,「文字列 (char*) のリスト」を作りたくなったとしよう．するとそのインタフェースはほとんど上と同じで，上記の int の部分を適宜 string や char* などに置き換えただけのものになるだろう．

- `typedef struct string_list { ... } * string_list_t`
- `string_list_t mk_string_list()`
- `void string_list_append(string_list_t` l, `char *` x`)`
- `char * string_list_get(string_list_t` l, `int` i`)`

そしておそらく，中身のコードもほとんど同じようなものになるのであろう．では次に,「整数ポインタ (int*) のリスト」「リストのリスト」. . . などといろいろなリストを作りたくなったら，そのたびに同じようなコードを書かなくてはいけなくなる．

ほとんど同じことをしたいのに，型の名前が違うとまったく別のコードを使わなくてはならないのだろうか? 結論を言うと，それは時と場合による．char *と整数 int は，32 bit CPU であれば，どちらも単なる 32 bit の整数として表されている．したがって実は，char *を受け取るはずの引数 (たとえば上記の x) のところに int を渡したり，その逆をやったとしても，そこで情報が失われるわけではない．もちろんそういうことをするとコンパイラがエラーメッセージを出す．しかし，キャストをする (例: (char *)10 のように書く) ことで，C のコンパイラを黙らせることができる．結果として以下のようなコードは「期待通りに」動作するのである．

```
main()
{
  string_list_t l = mk_string_list();
  /* 無理やり 20 をつっこむ */
  string_list_append(l, (char*)20);
  int x = (int)string_list_get(l, 0);
  /* ちゃんと 20 が帰ってくる */
  printf("I got back %d\n", x);
}
```

これが基本的なテクニックで，つまりは，ひとつの型に対するコード (この例で言えば，string_list_XXX) を書けば，あとは，引数を渡したり結果を受け取るときに適切なキャストを挟むことによって，その入れ物には「実際には何でも入れられる」のである．

しかし，プログラム上は型が明示されていたほうがわかりやすいし，常に型変換を入れて API を呼び出すのは面倒くさい．何よりも，自分自身がどのリストにどの型のデータが入っていたのかを間違えてしまう可能性がある．やはり int を入れるものと char*を入れるものは，プログラムの文面上別の型であることが望ましい．

そこで，よりベターなスタイルとしては，このようになる．

1. コンテナを定義する際には，入れるデータの型を void *型にしておく．つまり，中身のデータを void *型であると仮定して API を実装する．

2. 特定のデータを入れるコンテナ (たとえば int_list_t) を実装するには，まず int_list_t という型を**未知の構造体へのポインタ**として定義する．これは，int_list_t の例であれば，具体的には以下のように書く．

```
typedef struct int_list * int_list_t;
```

このようにした上で，int_list という構造体の中身は**どこにも定義しない!** これはエラーではなく，C コンパイラはともかく int_list_t という型が「何か (よくわからないもの) へのポインタ」であるということだけを理解する．

3. 各 API は中身を定義するが，それは適宜キャストを挟みながら，1. で定義した API を呼び出すだけのものにする．つまり，int_list_t の例で言えば，

```
void int_list_append(int_list_t l, int x)
{
  generic_list_append((generic_list_t)l, (void *)x);
}

int int_list_get(int_list_t l, int i)
{
  return (int)generic_list_get((generic_list_t)l, i);
}
```

のようになるであろう．ここで，generic_list_append などが，void *型に対して定義された API や型である．

こうしておけば，実際に int が入っているコンテナには常に int_list_t という型名がついているためプログラムも読みやすいし，API を呼ぶ際にいちいち (注意深く) キャストを挟む必要もなくなる．

最後の注意: このやり方は実際にどんな型に対しても通用するのかというと残念ながらそうではない．たとえば double 型のリストである double_list_t を，上記の要領で定義することはできない．もし書くとすれば以下のようになるだろう．

```
void double_list_append(double_list_t l, double x)
{
  generic_list_append((generic_list_t)l, (void *)x);
}

double double_list_get(double_list_t l, double i)
{
  return (double)generic_list_get((generic_list_t)l, i);
}
```

これをコンパイルすると，コンパイラが，(void *)x や (double)generic_list_get(...) に対して,「そんなキャストはできない」というエラーを出す.

しかし，そういうことよりもむしろ，それが禁止されている理由は，double (倍精度浮動小数点数) は 64 bit であって，32 bit である void *型へ，情報を失わずにキャストすることはできないから，と理解しておくほうが良いだろう．むしろ，int の場合に可能であったのは，所詮ポインタはすべて整数だからであり，キャストといっても機械語レベルでは何もしていないからである．同様の理由により，構造体 (構造体へのポインタではなく，構造体そのもの) と void *型の間のキャストも不可能である．つまりこの手法で定義できるコンテナは，**ポインタまたはポインタと同じサイズの整数** のコンテナと思っておくのが良い.

8.2 その他の言語の事情

同じコードをいろいろな型に対して再利用できること，それでいて，データの型が実行時に混同されないように，きちんと型の検査ができることは，プログラミング言語の設計上の目標である．これは言うほど簡単ではなく，言語によっていろいろな設計がなされている.

- C++は「テンプレート」をサポートしており，構造体を定義する際に一部の型を「パラメータ化」した形で定義することができる．たとえば,

  ```
  struct list<T> { ... };
  ```

  のように．ここで T が型パラメータであり，使う際に，list<int> のように具体的な型を代入して使う.

  C++の仕様は，実質的には「同じコードを型名だけを変えてコピーする」という作業をコンパイラが変わりにやってくれる，というのに似ており，実際いろいろな型のコンテナを実際に使うとその分コードサイズが膨れ上がるという問題がある.

- Java は Java 1.5 以前の仕様では，テンプレートをサポートしていなかった．慣習としてコンテナは，Object 型を入れるように定義するようになっており，Object と通常の class 型とのキャストが,「実行時のエラーチェック付」でサポートされている．型の体系としては Object 型が void* 型のようなものだと思えば C と似ている．しかし，データを取り出してもとの型にキャストをする際に，本当にその型のオブジェクトであるかどうかが実行時に検査される．つまり，間違えて型の取り違いを起こした際の挙動が C では，予測不能なメモリの破壊であるのに対し，Java では，より情報の多い処理系からのエラーメッセージである.

  int と Object の間のキャストは禁止されているので，int などのデータをそのままコンテナに入れることはできない．Integer オブジェクトなどのオブジェクトにして入れることになるが，そのオーバーヘッドは非常に大きい.

  Java 1.5 ではテンプレートに似た体系がしっかりと定義されている.

- ML という関数型言語があり，多相型がサポートされている．多相型は，C++のテンプレートなどが，もとはといえば参考にしているアイデアで,「同じコードがいろいろな型に対して

動く」コードを記述すること，その元できちんと型検査 (実行時型エラーが起きないことの保証) を行う．ML では，関数を書くと，引数や返り値の型を，処理系がある程度勝手に推論してくれ，かつ，適切な引数に対しては，実際にその型は何でも良いということまで推論してくれる．そのような関数は，引数に int やリストなど，ありとあらゆるデータをとることができる．コンテナを定義するのに必要なのはまさしくこのような関数である．詳しくは 4 年生のプログラミング言語の授業で出てくる．