情報通信工学実験 IB テキスト

国島丈生

第 1 章

構造体

1.1 構造体とは

構造体(structure) とは、複数の変数をひとまとまりにしたものである。構造体を構成する変数のことを**メンバー**(member) という。1 つの構造体に同じ名前のメンバーを含めることはできない。

例えば、平面座標上の点を表す構造体の宣言は次のようになる。

```
struct point {
    int x;
    int y;
};
```

メンバーを宣言する順番に制限はない。この例で、変数 y を x より先に宣言しても、プログラム上での扱いは変わらない。

メンバーの型には制限はない。つまり、いろいろな型のメンバーが 1 つの構造体に含まれていてもよい。後述するように、ポインタ型の変数をメンバーにしてもよい。

tgif などのドローソフトには、いくつかの図形をグループ化する機能がある。グループ化された図形たちは、あたかも 1 つの図形であるかのように拡大縮小したり、コピー&ペーストしたりすることができる。構造体は、この「グループ化」の機能に近い。

課題 1.1　個人データを表す構造体を作成せよ。個人データの項目は配布された課題にある通りとする。

1.2 型としての構造体

いったん構造体を定義すると、以降、その構造体は型として扱える。つまり int や char などと同じように扱うことができる。例えば、先の構造体 point を宣言すると、次のように、struct point 型の変数を宣言したり利用したりできる。

```
/* struct point 型の変数 pt の宣言 */
struct point pt;

/* struct point 型の変数 maxpt の宣言、および初期化。
   point の宣言に従って、値とメンバーが対応される。この例では、
   x と 320 が、 y と 200 がそれぞれ対応される。
```

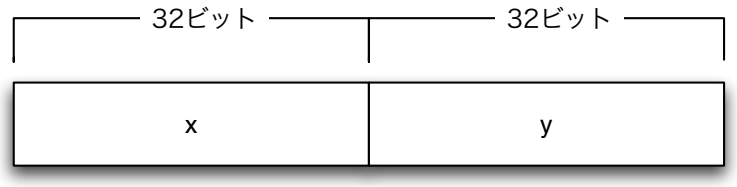


図 1.1　構造体 point のメモリ上の配置

```
 */
struct point maxpt = {320, 200};

/* 構造体の宣言と変数宣言を一緒にしてもよい */
struct float_point {
  float x;
  float y;
} float_pt;
```

構造体と変数の関係を述べる前に、変数と型の正体について述べておく必要があるだろう。

変数 (variable) とはいわば「入れ物」であり、メモリ上に確保された一定サイズの領域である。その中に入れる (格納する) ことのできる値 (value) の制限が**型** (type) である[*1]。例えば `int i;` と宣言すると、(多くの場合)32 ビット分のメモリ領域が自動的に確保され[*2]、そこに `i` という名前がつけられる。これが変数 `i` の正体である。そして、変数 `i` には `int` 型の値、すなわち整数しか格納できない。

では、構造体はどのようなものとしてメモリ上に配置されているのだろうか。実は、メンバーの領域が連続して配置されたようなメモリ領域になっている。ただし、構造体独自の領域がある場合もある。

例えば、変数 `pt` は図 1.1 のようにメモリ上に配置される。`struct point` 型の変数 `pt` を宣言すると、`struct point` 型に相当するメモリ領域 (この場合は 32 ビット +32 ビット=64 ビット) が自動的に確保され、そこに `pt` という名前がつけられる。そしてこの領域には、`struct point` 型の値しか格納できない。`struct point` 型の値とは、具体的には 2 つの整数の組であり、例えば (320, 200) のようなものである。

1.3　メンバーの値を参照するには

構造体のメンバーの値を参照するにはどうしたらよいのだろうか。例えば、平面上の点の x 座標の値を得るにはどうしたらよいのだろうか。

それには**構造体メンバー演算子** (structure member operator) “`.`” を用いる。これは二項演算子であり、各演算数は次の通りでなければならない。

- 第 1 演算数は構造体型の変数や式[*3]
- 第 2 演算数はメンバー名

```
struct point maxpt = {320, 200};
```

[*1] 型は、数学で言うところの定義域 (domain) と似た概念である。

[*2] 正確には、コンパイル時に、32 ビットのメモリ領域がプログラム中に確保される。詳しくは講義「コンパイラ」で扱う。

[*3] ただし `{320, 200}.x` というような記法は許されない。

```
/* メンバー x の値の参照 */
printf("%d\n", maxpt.x);

/* メンバー y への代入 */
maxpt.y = 240;
```

1.4 構造体の配列

構造体は型として扱えるのだから、もちろん構造体の配列も許される。

```
struct point pts[100];

/* 要素数 3 の配列 pts1 の宣言と初期化 */
struct point pts1[] =
  {{320, 200},
   {150, 500},
   {0, 0}};

/* pts1 の各要素は struct point 型の値 */
pts1[1].y = 240;
```

1.5 構造体へのポインタ参照

1.5.1 ポインタについてのまとめ直し

構造体は型として扱えるのだから、もちろん構造体へのポインタ参照も許される。この話に入る前に、ポインタについて簡単にまとめ直しておく。

先に述べたように、変数の正体はメモリ領域である。ということは、変数と、(変数の正体である) メモリ領域の先頭番地は 1 対 1 に対応するはずである[*4]。

そこで、C では以下の 2 つの単項演算子を用意した。

- **アドレス演算子** (address operator) `&`：変数名からメモリ領域の先頭番地を得る演算子
- **間接演算子** (indirection operator) `*`：メモリ領域の先頭番地から変数の値を得る演算子[*5]

また、型 `X` に対して、`X` 型のメモリ領域の先頭番地を表す型 `X*`を用意した。例えば `int *pi;` と書くと、変数 `pi` には `int` 型のメモリ領域の先頭番地しか格納できない[*6](図 1.2)。

[*4] メモリ領域の末尾番地も扱わないと不公平じゃないか、というように考える人がいるかもしれないが、末尾番地は「先頭番地 + 変数の型の大きさ」で計算できる、ということでお許しいただこう。

[*5] ただし、C コンパイラは、本当に「先頭」かどうかの保証はしてくれない。プログラマが責任を持って、必ず「先頭」であるようにプログラムを書かなければならない。皆さんご承知の通り、これを間違えると、segmentation fault や bus error を引き起こす。

[*6] ただし、実は C では「先頭番地」でなくても格納できてしまう。これも segmentation fault や bus error の原因となる。

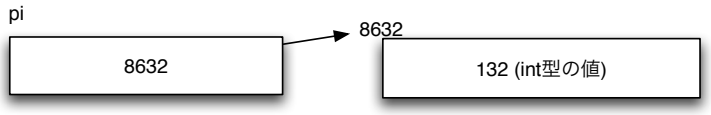


図 1.2　int *pi;

1.5.2　構造体に対するポインタ

前節のポインタに関する説明は、構造体にもそのままあてはまる。以下の 2 つの変数宣言はまったく異なることに注意せよ。

- struct point pt; … 変数 pt は struct point 型の値を格納できるだけの領域 (64 ビット) を持つ。格納できる値は (320, 200) のような 2 つの整数の組である。
- struct point *p; … 変数 p は番地を格納できる領域 (現在は多くの場合 32 ビット) を持つ。格納できる値は struct point 型の値の格納領域の**先頭番地**である。

これまで呪文のように書いてきた以下のプログラムコードも、これまでの説明を踏まえると、体系的に理解することができる。

- struct point *p = (struct point *)NULL;
  C では、NULL は整数 0 の別名である。すると、このコードの意味は「番地を格納する 32 ビットのメモリ領域 p を確保し、そこに 0 を代入する」となる。ただし、0 は通常 int 型であり、変数 p の型とは異なる。そこで、(struct point *)NULL というように、明示的に 0 の型を変換 (キャスト, cast) してから代入を行っている。
- struct point *p = (struct point *)malloc(sizeof(struct point));
  変数 p の領域を確保するところまでは上の例と同じ。
  malloc() 関数は次のような動きをする。
  – プログラム実行時[*7]に、(引数で与えられた整数値) バイトの領域を確保
  – 確保した領域の先頭番地を、void *型の値として返す
  – 確保した領域の初期値は未定義 (何が入っているか分からない)
  sizeof は (なんと!) 単項演算子で、しかも型名を演算数にとる。結果は、その型の値を格納するのに必要なバイト数である。したがって sizeof(struct point) = 8(8 バイト=64 ビット) である。
  まとめると、malloc() により sizeof(struct point) バイトの領域を動的に確保し、その先頭番地を明示的に struct point *型に変換してから p に代入していることになる (図 1.3)。ただし malloc() で確保した領域の初期値は分からないので、プログラマが自分で初期化しておかなければならない。

1.5.3　ポインタ参照した構造体のメンバー

図 1.3 において、動的に確保した構造体のメンバー x の値を読み出すにはどうしたらよいだろうか。

8236 番地から始まる構造体は、*p で得られる。したがって、この構造体のメンバー x は (*p).x で得られる。ただし、二項演算子 . は単項演算子*よりも優先順位が高いので、カッコが必要になる。

[*7] ヒープ領域と呼ばれる特別な場所に確保する。詳細は「コンパイラ」の講義で述べる。

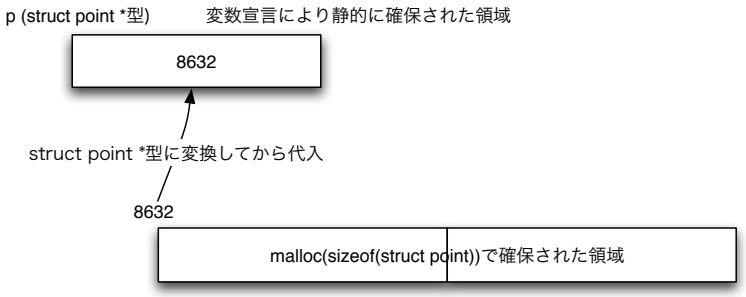


図 1.3 領域の動的確保

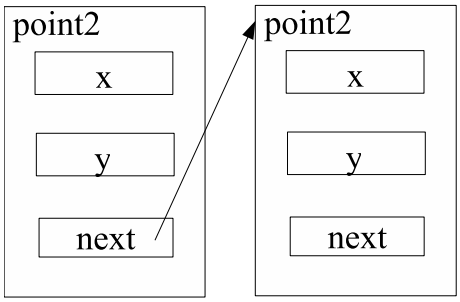


図 1.4 point2 型の構造体

C のプログラムでは、この種のコードが頻繁に出てくるので、カッコが多くなると煩わしい[*8]。そこで、C には、(*p).x の略記法が用意してある。p -> x である。この 2 つの式はまったく等価である。

なお、演算子->は左結合であることに注意されたい。すなわち、a -> b -> c は (a -> b) -> c の意味になる。

1.6 ポインタ型を含む構造体

ここまでの説明で述べた通り、ポインタ変数とは単に「番地 (という整数値) しか格納できない変数」である。したがって、ポインタ変数をメンバーとするような構造体も許されるはずである。例として、次のような構造体を考える。

```
struct point2 {
  int x;
  int y;
  struct point2 *next;
};

struct point2 *pp;
```

これは図 1.4 のような構造体を表している。メンバー next が、別の struct point2 型の構造体を指し示すポインタになっている。左側の構造体の先頭番地が変数 pp に格納されているとすると、右側の構造体の先頭番地は p -> next である。したがって、右側の構造体のメンバー x の値を得るには (pp -> next) -> x、もしくは pp -> next -> x とすればよい[*9]。

[*8] Lisp のように…

[*9] ->は左結合の演算子であることに注意。

1.7　課題: 文字列操作

課題 1.2　C 言語には、文字列を操作するライブラリ関数として strcpy, strcmp, strcat などがある。これらの関数について、以下の項目をマニュアル等を用いて調べよ。

- 引数の数・型、返り値の型
- 関数の動作の概要、引数や返り値の意味

課題 1.3　課題 1.7 で調べた内容を基に、strcpy, strcmp, strcat と全く同じ振舞いをする関数を実装し、動作を確認せよ。関数名はそれぞれ my_strcpy, my_strcmp, my_strcat とせよ。

ここでいう「振舞い」とは、関数の動作だけではなく、引数の数・型、返り値の型、それらの意味なども含まれる。つまり、strcpy の代わりに my_strcpy を用いても全く同じ動作をしなければならない。

動作の確認には、my_strcpy, my_strcmp, my_strcat を使用して main() を実装し、その動作が予想通りになっているかどうか調べれば良い。

1.8　課題: 住所録の作成

1.8.1　仕様

作成する電話番号簿の仕様は次の通りとする。

- 個人データは、以下の 3 個の項目から構成されることとする。
  1. 氏名（アルファベット）：char[256]（文字列）
  2. 電話番号：char[16]（文字列）
  3. 学籍番号：int
- 最大 5 名分のデータを管理できること。
- 新規追加が可能であること。
- 指定されたデータの削除が可能であること。削除されたデータ項目は次の仕様に基づいて「登録がない」状態とすること。
- 登録がないデータ項目は、氏名および電話番号についてはヌル文字列、学籍番号については 0 が格納されていることとする。
- 上記の仕様を用いて、一度削除されたデータ項目は以降の新規追加処理で再利用可能であること。

1.8.2　課題

構造体を用いない実装

課題 1.4　1.8.1 節で述べた個人データを管理する住所録を、名前、電話番号、学籍番号を表す 3 つの配列 name, phone, number を用いて実装せよ[*10]。

1. (データの新規追加) 引数を順に名前、電話番号、学籍番号とするようなデータ項目を住所録に追加する関数

```
void add(char *, char *, int);
```

[*10] name, phone, number それぞれの 0 番目の要素で 0 番目の個人データを表現するというような実装になる。住所録の表の各列 (名前、電話番号、学籍番号) をそれぞれ 1 つの配列で実装するようなイメージ。

2. (データ項目の削除) 引数で指定されたデータ項目と合致する個人データを住所録から削除する関数

```
void delByName(char *);
void delByPhone(char *);
void delByNumber(int);
```

例えば `delByName()` は、引数で指定された文字列を名前とするような個人データを住所録から削除する。`delByPhone()`, `delByNumber()` も同様である。

3. (データの検索) 引数で指定されたデータ項目を住所録から検索する関数

```
void findByName(char *);
void findByPhone(char *);
void findByNumber(int);
```

例えば `findByName()` は、引数で指定された文字列を名前とするようなデータ項目を住所録から検索する。`findByPhone()`, `findByNumber()` も同様である。
これらの関数は、合致するすべての個人データを表示するものとする。

4. (データの表示) 住所録に登録されている個人データをすべて表示する関数

```
void printAll();
```

未使用の領域も出力すること。

を作成せよ。

- 引数の意味に充分注意すること。例年、引数の意味を誤解して実装を行う学生が少なからず見受けられる。
- 文字列の一致判定には `strcmp` を用いるとよい。
- データを格納する配列 `name`、`phone`、`number` はグローバル変数（大域変数）で確保してよい。

課題 1.5　課題 1.4 で作成した関数を用いて住所録データを操作する `main()` 関数を作成せよ。その実行結果も示せ。

- 今回の実験の主眼は `main()` ではなく、課題 1.4(および課題 1.7) である。したがってデータの入出力に凝る必要は必ずしもない。例年、ユーザとの対話的入出力を実装してくる学生が多い[*11]が、そこまでは要求しない。
- ただし、課題 1.4 で実装した関数が正しく動作することを確認するように実装を行うこと。すなわち、最低限、`main()` の中ですべての関数を使うようにしなければならない。

構造体を用いた実装

課題 1.6　1.8.1 で述べた個人データを表す構造体の宣言を書け。構造体のメンバー名は自由に決めてよい。

課題 1.7　課題 1.4 で作成したものと同一（機能だけではなく引数の個数、型も同一）の関数群を構造体の配列を用いて実装せよ。ここで用いる構造体は、課題 1.6 で作成した構造体を用いること。なお、データを格納する構造体の配列はグローバル変数（大域変数）で確保してよい。

課題 1.8　課題 1.7 で作成した関数を用いて住所録データを操作する `main()` 関数を作成せよ。その実行結果も示せ。

[*11] 出所は 1 つだと思われるが…

1.9　課題：まとめ

課題 1.9　今回実装したプログラムについて、改良する点があるとすればどのようなことが考えられるか。また、その改良を実装する場合の実装方針、問題点、難しいと思われる点などについて考察せよ。

第 2 章

リスト

2.1 リストの数学的定義

数学的には、「ある同一の型の要素を 0 個以上並べたもの」を**リスト**という[*1]。記法は文献によってまちまちであるが、この実験では $[a_1, a_2, \cdots, a_n]$ というように、要素をコンマで区切り、角括弧で囲って書くことにする[*2]。このとき、リストに含まれる要素数をリストの長さという。また、要素が一つもないリストを**空リスト**といい、[] と書く[*3]。

リストの要素は、リスト内の位置によって全順序 (total order) の関係をつけることができる。つまり、リスト内の任意の 2 要素を取ってくると、それらの間にリスト内の位置による順序関係を決めることができる。なお、ここでいう順序関係は、要素間の大小とは関係ないことに注意すること。たとえば、整数型の要素からなるリスト $[1, 3, 2]$ を考えると、要素間に次のような順序関係をつけることができる。

- リスト内の位置による順序: 1(第 1 要素), 3(第 2 要素), 2(第 3 要素)
- 要素の大小による順序: 1(第 1 要素), 2(第 3 要素), 3(第 2 要素)

2.2 リスト内の基本操作

リストに限らず、この実験で扱うような論理的なデータ構造では、それに対する基本的な操作群を考えることができる。これをうまく定めれば、そのデータ構造に対するあらゆる操作は、操作群中の操作のみを用いて構成することができる。

リストに対する基本操作群の一例として、図 2.1 のようなものを考えることができる。なお、このような基本操作群の定め方は一通りとは限らない。たとえばプログラミング言語 ML では、図 2.2 のようなものを基本操作群としている。

2.3 リストの実装

これまで述べてきたリストをプログラミング言語で実装する方法はいくつかある。この章では、実装方法のうちのいくつかについて概略を述べる。

プログラミング言語の中には、リストをライブラリとしてすでに持っているものもある。Lisp, ML, Prolog, Java などはその例である。一方 C はリストをライブラリとして持っていないので、リストを使いたい場合は自分で実装を行

[*1] 「同一の型」であるかどうかは定義による。この講義ではこの定義にしたがうものとする。

[*2] この記法は「計算機言語 I」で扱う ML にならっている。

[*3] Lisp や ML のような関数型言語では “nil” と書くことも多い。

$FIRST(L)$　リスト L の先頭の要素の位置を返す。
$END(L)$　リスト L の末尾の要素の位置を返す。
$INSERT(x, p, L)$　要素 x をリスト L の位置 p に挿入する。結果として、L の p 番目以降の要素はすべて 1 つずつ後ろに移動することになる。リスト L に位置 p がない場合は、結果は定義されない。
$LOCATE(x, L)$　リスト L 内の要素で、x と同じ値を持つものの位置を返す。該当する要素が 2 つ以上ある場合は該当する最初の要素の位置を返す。また該当する要素がないときの結果は定義されない。
$RETRIEVE(p, L)$　位置 p にあるリスト L の要素を返す。L に位置 p がない場合の結果は定義されない。
$DELETE(p, L)$　位置 p にあるリスト L の要素を削除する。L に位置 p がない場合の結果は定義されない。
$NEXT(p, L), PREVIEOUS(p, L)$　位置 p にあるリスト L の要素の一つ次の要素の位置、一つ前の要素の位置をそれぞれ返す。$NEXT(END(L), L)$ や $PREVIOUS(FIRST(L), L)$ の値は定義されない。また L に位置 p がない場合は $NEXT$, $PREVIOUS$ ともに結果は定義されない。
$MAKENULL()$　空リストを返す。
$PRINTLIST(L)$　リスト L の要素を順にすべて印字する。

図 2.1　リストの基本操作群

nil　空リスト。
$cons(x, L)$　リスト L の先頭に要素 x を加えたリストを返す。
$hd(L)$　リスト L の先頭の要素を返す。
$tl(L)$　リスト L から先頭の要素を取り除いたリストを返す。
$L@M$　リスト L の後ろにリスト M を連接したリストを返す。

図 2.2　ML におけるリストの基本操作群

う必要がある。

2.3.1　配列による実装

もっとも簡単な実装は配列を用いたものである。図 2.3 のように、配列内の連続したセルの中にリストの要素を順に格納する。

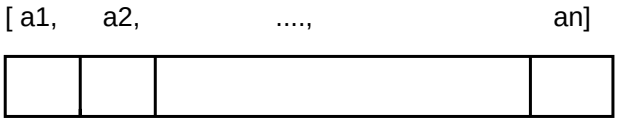


図 2.3　配列によるリストの実装

この方法は実装が容易であり、また、リストの各要素を調べたり最後に新しい要素を尽け加える操作は簡単に実装できる。しかし、配列の大きさを越える長さのリストは扱うことができない。また、リストの途中に要素を挿入する場合、それ以降の要素をすべて 1 つずつずらして、新しく挿入する要素のための領域を確保しなければならない。同様に、途中の要素を削除する場合も、後ろの要素を前に詰めて、すきまを埋める必要がある。

2.3.2 構造体によるリストの実装 (1)

構造体を用いてリストを実装することもできる。この場合、各要素を表すデータ構造と、次の要素へのポインタとを構造体 (**セル**) として構成し、リストの出現順につないでいくことになる。図 2.4 に実装例を示す。この実装のことを連結リストということがある。なお、この図のように、連結リストの先頭に特別なセル *header* を設けておくと都合のよいことが多い。

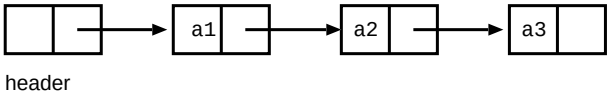


図 2.4 構造体によるリストの実装

この方法では、リストの要素の増減に合わせて記憶領域を動的に確保することができ、リストの長さに制限がない。また、リストを格納するために連続した記憶領域を使わなくてよいので、要素の挿入や削除の実装が簡単になる。そのかわり、ポインタのための領域を余分に使用する、要素の検索は先頭の要素から順にポインタをたどっていく必要がある、などの欠点がある。

今回の実験では、この方式による実装を行う。

2.3.3 構造体によるリストの実装 (2)

Lisp や ML などのプログラミング言語では、$[[1,2,3],[4,5],nil]$ のようにリストを入れ子にすることができる。このようなリストを許す場合には、ここまでに述べたものとは違う実装が必要になる。ML におけるリストの実装を図 2.5 に示す。

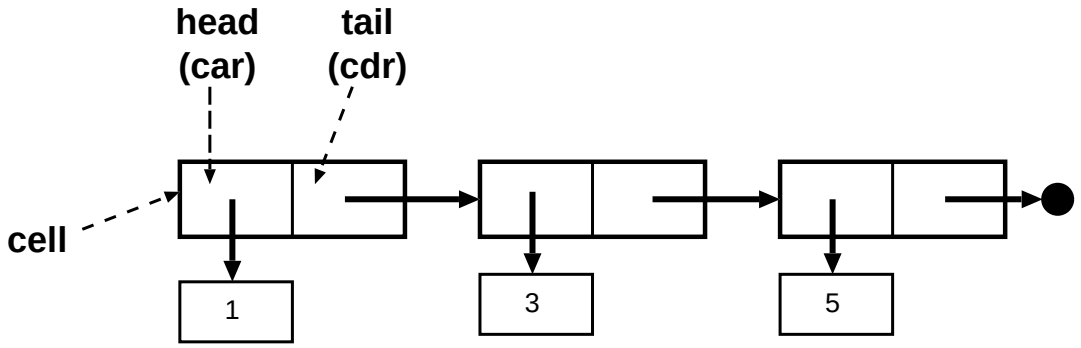


図 2.5 構造体によるリストの実装 (2)

基本的には構造体を使うが、各要素へのポインタと、次の要素へのポインタとをセルとして構成するところが前節の実装と大きく異なる点である[*4]。

2.4 構造体を指し示すポインタ

以下の練習問題は、実験レポートに含める必要はない。

[*4] この実装については、「計算機言語 I」で詳しく述べる予定である。

【練習問題 1】 個人データを保持する構造体として以下のようなものを作成した (前回の実験を思い出せ)。この宣言に、構造体 personal_data へのポインタを保持するメンバ next を追加せよ。

```
struct personal_data {
  char name[256];
  char phone[16];
  int number;
};
```

【練習問題 2】 struct personal_data 型の構造体が 2 つ、メモリ上に置かれており、それらの先頭番地がそれぞれ struct personal_data *型の変数 a, b に格納されているものとする。

```
struct personal_data *a, *b;
/* 実際のプログラムでは、malloc() を用いてメモリ領域を動的に確保・
初期化し、それらの先頭番地を a, b に代入するという作業が必要であ
る。ここでは、すでにこの作業は行われているものとする */
```

さて、a (で指し示されている構造体) の次に b (で指し示されている構造体) をつなぎたい。どのようなコードを書けばよいか、示せ。

【練習問題 3】 練習問題 2 に続き、「a(で指し示される構造体) の次の構造体の氏名」を printf() で表示させたい。どのようなコードを書けばよいか、示せ。(変数 b は使用せずに書くこと)

2.5 課題: 連結リストによる「リスト」の実装

2.5.1 仕様、実装上の注意

リストは C 言語には元々ないデータ構造なので、プログラム中でリストを扱うためには、C 言語に用意されているデータ構造で実装を行う必要がある。実装方法はいくつかあるが、ここでは、連結リストを採用することとする。細かい仕様は以下の通りである。

- リストの各要素に構造体 (以下「セル」と呼ぶ) を一つ対応させ、リスト中で並んでいる順にセルをポインタで接続するものとする (テキストの図 2.4 参照)。
- セルデータは int 型のデータ 1 個だけとする。
- 次の要素が存在しない場合には、対応するセルでは、次のセルへのポインタを NULL とする。すなわち、次のセルとして NULL を保持するセルは、リストの終端となる。
- 各要素を表すセルの他に、連結リストの先頭にセルを 1 つ置く (これを以下「ダミーセル」と呼ぶ)。リスト全体を C 言語中で指し示す必要がある場合には、ダミーセルへのポインタで代用する。例えば、空リストは、ダミーセル 1 つだけからなる連結リストで表現される。
- 操作関数は、以下の課題で述べる 11 個である。各関数の仕様はそれぞれの課題で述べるが、いずれもテキストの図 2.1 にならったものとしている。

なお、これらの関数はすべてリスト全体を指し示す引数を持つように統一している。そのため、一部の関数では、引数をすべて使用しなくても実装できることがある。

- この課題で作成した関数群は次回の実験で利用するため、構造体宣言、関数のプロトタイプ宣言等はヘッダファイル（例えば `list.h`）に、操作関数群と操作プログラムは別ファイル（例えば `list.c` と `list-test.c`）とし、コンパイルには `make` を用いる。

一部の課題では解答例を示している。この解答例では、セルの構造体のメンバとして次のセルへのポインタを保持する変数名を `next` と仮定している。もちろん、レポートに添付するプログラムは、この解答例の通りである必要はない。

2.5.2 課題

課題 2.1 `int` 型のデータを 1 つ持つ連結リストのセルを表す構造体 (`struct cell`) を記述せよ。

課題 2.2 空リストを作成する関数

```
struct cell *makeNullList(void);
```

を作成せよ。返り値は、ダミーセルへのポインタである。

ダミーセルは `malloc` を用いて動的に領域を確保すること。また、動的確保が成功したかどうかを必ず確認し、失敗した場合にはその旨を表示してプログラムを終了させるようにすること[*5]。また、ダミーセルの次のセルへのポインタの初期値は `NULL` とし, 次がないことを表しておく。

【解答例】

```
/*
 *  Make null list and return pointer to list.
 */
struct cell *makeNullList(void)
{
  struct cell *list;  /* pointer to new list */

  /* allocate memory for new list entry */
  list = (struct cell*)malloc(sizeof(struct cell));
  /* check the allocation succeeded or failed */
  if (list == (struct cell*)NULL) {
    fprintf(stderr, "makeNullList: can not allocate memory for new list.\n");
    exit(1);
  }

  /* set next cell's position to NULL */
  list->next = (struct cell*)NULL;

  return list;  /* return pointer to list */
}
```

課題 2.3 指定されたセルの次のセルへのポインタを返す関数 `nextCell` を作成せよ。

```
struct cell *nextCell(struct cell *, struct cell *);
```

第 1 引数は指定するセルへのポインタである。また、第 2 引数は対象となるリスト全体を表す。すなわち、操作対象としている連結リストのダミーセルへのポインタである。

[*5] メモリの動的確保に失敗するのは、異常に重いプログラムが実行されているなど、コンピュータ上でのプログラムの実行に重大な問題が発生している場合が多い。

【解答例】[a]

```
/*
 *  Return pointer to the next cell.
 */
struct cell *nextCell(struct cell *target, struct cell *list)
{
  struct cell *next = (struct cell*)NULL;


  if (target != (struct cell*)NULL) {
    next = target->next;
  }

  return next;
}
```

[a] 仕様上はリストへのポインタを引数としてとるが、この解答例ではこの引数を利用していない。

課題 2.4　連結リストの先頭のセル、終端のセルへのポインタを返す関数

```
struct cell *firstCell(struct cell *);
struct cell *endCell(struct cell *);
```

を作成せよ。いずれの関数もダミーセルへのポインタを引数とする。連結リストが空リストを表している場合には、いずれの関数も NULL を返すものとする。

課題 2.5　指定されたセルの前のセルへのポインタを返す関数 previousCell を作成せよ[*6] 。

```
struct cell *previousCell(struct cell*, struct cell*);
```

第 1 引数は指定するセルへのポインタ、第 2 引数は対象となる連結リストのダミーセルへのポインタである。

課題 2.6　新たにセルを作成し、指定された位置に挿入する関数 insertCell を作成せよ。

```
struct cell *insertCell(int, struct cell *, struct cell *);
```

第 1 引数はデータの値、第 2 引数は挿入位置となるセルへのポインタ、第 3 引数は対象となる連結リストのダミーセルへのポインタである。新たに作成されたセルは、第 2 引数のセルの前に挿入することとする。関数の返り値は、挿入したセルへのポインタである。

連結リストの最後尾にセルを追加するときは、位置 (第 2 引数) として NULL を指定する[*7]。また、新規に作成するセルの格納領域は malloc を用いて動的に確保する。

課題 2.7　指定されたセルを指定されたリストから削除する関数 deleteCell を作成せよ。削除したセルのメモリは解放すること。

```
struct cell *deleteCell(struct cell*, struct cell*);
```

[*6] 本課題の仕様では、セルは直前のセルを示すポインタを保持していないため、先頭のセルから順にたどる必要がある。

[*7] nextCell(endCell(list), list) が NULL であるため。

第 1 引数は指定するセルへのポインタ、第 2 引数は対象となる連結リストのダミーセルへのポインタである。返り値は、削除されたセルの次に位置していたセルへのポインタである。

課題 2.8 指定されたセルのデータを返す関数 retrieveCell を作成せよ。ただし、指定されたポインタが NULL であった場合には, 0 を返すこととする。

```
int retrieveCell(struct cell*, struct cell*);
```

第 1 引数はデータを取得したいセルへのポインタ、第 2 引数は対象となる連結リストのダミーセルへのポインタである。

課題 2.9 指定されたデータを値として持つセルへのポインタを返す関数 locateCell を作成せよ。ただし、該当するセルがない場合には NULL を返すこと。また、同じ値をもつセルが複数ある場合にはそのうちのいずれか 1 つのみを返すものとする。

```
struct cell *locateCell(int, struct cell*);
```

第 1 引数は指定するデータの値、第 2 引数は対象となる連結リストのダミーセルへのポインタである。

【解答例】

```
/*
 *  return position of the cell has data equals to specified as iData
 */
struct cell *locateCell(int data, struct cell *list)
{
  struct cell *current;  /* current cell's pointer */


  /* set pointer to current cell as pointer to first cell */
  current = firstCell(list);

  /* while current cell's data is not data, pursue the next cell */
  while ((current != (struct cell*)NULL)
       && (retrieveCell(current, list) != data)) {
    current = nextCell(current, list);
  }

  return current;  /* return end cell's pointer */
}
```

課題 2.10 先に作成した関数 deleteCell を用いて、指定された連結リスト中のすべてのセルを削除する関数 deleteList を作成せよ。ダミーセルも含めて削除すること。

```
void deleteList(struct cell*);
```

第 1 引数は対象となる連結リストのダミーセルへのポインタである。

課題 2.11 指定されたリストの内容を表示する関数 printList を作成せよ。ただし、表示はリストの先頭から順に [1, 2, 3, 4, 5] の形で表示することとする。

```
void printList(struct cell*);
```

第 1 引数は対象となる連結リストのダミーセルへのポインタである。

課題 2.12 実装を行った連結リスト操作関数群を用いて連結リストを操作する main() を実装せよ。連結リスト操作関数群が正しく動作していることが確かめられるように、基本関数群の使い方を工夫すること。

2.6　課題：まとめ

課題 2.13　今回実装したプログラムについて、改良する点があるとすればどのようなことが考えられるか。また、その改良を実装する場合の実装方針、問題点、難しいと思われる点などについて考察せよ。

第 3 章

スタック、キュー

リストの特殊な場合として重要なものに、スタックとキュー (待ち行列) がある。これらはリストとは独立した概念として計算機科学の分野で登場することも多い。

3.1 スタック

3.1.1 定義と基本操作群

リストの中で、挿入や削除がいつも一方の端からしかできないものを**スタック**という[*1]。追加や削除をする側の端をスタックの**先頭**、もう一方の端をスタックの**底**という。スタックでは、あとで追加した要素のほうが先に取り出されるという性質があり、このことからスタックを LIFO(Last-In First-Out) と呼ぶこともある。

スタックでは追加や削除が一方の端からしかできない。また、先頭以外のスタック中の要素を参照することもできない。したがって、スタックの基本操作群はリストよりも簡単になる。一例を図 3.1 に示す。

$MAKENULL(S)$　スタック S を空にする。
$TOP(S)$　スタック S の先頭要素を返す。返した要素はスタックから降ろさない。
$POP(S)$　スタック S の先頭要素をスタックから降ろし (削除)、降ろした要素の値を返す[a]。
$PUSH(x, S)$　要素 x をスタック S の先頭に積む (追加)。
$EMPTY(S)$　スタック S が空のとき真、そうでないとき偽を返す。

[a] 実装によっては、スタックから降ろした要素を返り値として返さないこともある。

図 3.1　スタックの基本操作群

3.1.2 リストによる実装

リストの基本操作群を用いてスタックの基本操作群を実装することを考える。このとき、スタックの先頭をリストのどちら側の端にするかを決めておかなければならない。リストによる実装の場合は、実装のしやすいほうにしておけばよい。

ここではリストの先頭をスタックの先頭にすることとしよう。すると、たとえば $TOP(S)$ はリストの先頭の要素を返せばよいのだから、$RETRIEVE(FIRST(S), S)$ と書くことができる。また $PUSH(x, S)$ はリストの先頭に要素

[*1] 本を机の上に積んだ状態をイメージしてもらうとよい。

x を挿入すればよいのだから、$INSERT(x, FIRST(S), S)$ と書くことができる。

このようにして実装されたスタックは、内部的な実装はリストであるが、外部から基本操作群によって利用する限り、内部の実装を利用者が知る必要はない。このように、基本操作群で内部の実装を隠すという考え方を「情報隠蔽」もしくは「抽象データ型」と呼ぶことがある。オブジェクト指向プログラミング言語の「クラス」は、この考え方を発展させたものである。詳しくは「計算機言語 II」で述べる。

3.1.3　配列による実装

スタックを配列で実装することもできる。多くの場合、配列の要素数は固定であるから、スタックに積むことのできる最大要素数も固定されることになる。また配列では、添字の大きいほうをスタックの先頭にしておくと実装しやすくなる。添字の小さいほうをスタックの先頭にすると、一般には要素の挿入・削除のたびに配列の詰め直しをしなければならない。

たとえば、スタックを実装する配列を $elements[0 \cdots 99]$、スタックに現在入っている要素数を $length$ とすると、$TOP(S)$ は $elements[length - 1]$ を返すような関数として実装できる。また $PUSH(x, S)$ は次のような関数として実装できる。

```
push(x, S) {
  if (length == 100) {
     もう追加できない;
  } else {
     elements[length++] = x;
  }
}
```

3.2　キュー (待ち行列)

3.2.1　定義と基本操作群

リストの中で、挿入が一方の端から、削除が反対の端からしかできないものを**キュー** *(**待ち行列**)* という。キューでは、先に追加された要素ほど先に取り出されるという性質があり、このことから FIFO (First-In First-Out) と呼ばれることもある。

スタックの場合と同様、待ち行列の基本操作群はリストより簡単になる。一例を図 3.2 に示す。

$MAKENULL(Q)$　待ち行列 Q を空にする。
$FRONT(Q)$　待ち行列 Q の先頭要素を返す。返した要素は待ち行列から取り除かない。
$ENQUEUE(x, Q)$　要素 x を待ち行列 Q の最後に追加する。
$DEQUEUE(Q)$　待ち行列 Q の先頭要素を Q から削除し、その要素を返す。
$EMPTY(Q)$　待ち行列 Q が空なら真、そうでなければ偽を返す。

図 3.2　待ち行列の基本操作群

3.2.2 リストによる実装

スタックの場合と同様、リストのどちらの端を待ち行列の先頭・末尾に対応させるかは決めておく必要がある。リストの基本操作群の種類を考慮しながら、実装のしやすいほうに定めればよい。

ここではリストの先頭を待ち行列の先頭、リストの末尾を待ち行列の末尾にそれぞれ対応させることにする。すると、たとえば $FRONT(Q)$ はリストの先頭の要素の値を返せばよいのだから $RETRIEVE(FIRST(Q), Q)$ と書くことができる。また $ENQUEUE(x, Q)$ はリストの末尾に要素 x を加えればよいのだから、$INSERT(x, END(Q), Q)$ とすればよい。

3.2.3 配列による実装

待ち行列を配列で実装することもできるが、スタックの場合と異なり、少し工夫が必要になる。

配列の先頭を待ち行列の先頭、配列の末尾を待ち行列の末尾にそれぞれ対応させ、$ENQUEUE(x, Q)$ を配列の末尾に要素を加える、$DEQUEUE(Q)$ を配列の先頭から要素を削除する、という実装方針はすぐ思いつくだろう。しかし、一般に配列の要素数は固定であるため、上記の方針だけでは、何回か追加を繰り返すと配列の最大要素数に達し、それ以上追加ができなくなってしまう。

これを避ける一つの方法として、論理的に配列が循環しているように実装を行う方法がある。つまり、配列の末尾の要素の次がその配列の先頭になるように考えるわけである。たとえば、待ち行列を実装する配列を $elements[0 \cdots 99]$ とすると、$elements[99]$ の次の要素は $elements[0]$ であるように実装を行う。待ち行列の先頭の添字を表す変数を $front$、長さを表す変数を $length$ とすると、$ENQUEUE(x, Q)$ は次のように実装できる。

```
ENQUEUE(x, Q) {
   if (length == 100) {
       もう追加できない;
   } else {
       elements[(front + length) % 100] = x;
       length ++;
   }
}
```

3.3 スタックの実装

3.3.1 プログラムの仕様

- 内部構造として連結リストを用いる。
- 先週作成したリスト操作ライブラリをできる限り活用する。
- スタックの先頭を連結リストのどちら側に対応させるかは各自で決めること。
- 扱うデータは `int` 型のデータとする。
- 操作関数は、スタックの新規作成 (MAKENULL)、先頭データの取得 (TOP)、先頭セルの取り出し (POP)、データの新規追加 (PUSH)、空スタックの検査 (EMPTY) にスタック全体の削除を加えた計 6 関数から構成する。

また、実使用時に利用しやすいように、構造体宣言、関数のプロトタイプ宣言等はヘッダファイル（例えば stack.h）に、操作関数群と操作プログラムは別ファイル（例えば stack.c と stack-test.c）とし、コンパイルには make を用いるようにすること。

なお、前回作成したリスト操作ライブラリを利用することから、今回の課題において構造体メンバへの直接アクセスは不要となるはずである。課題回答中にこのような操作が必要となった場合には、リスト操作関数のいずれかで代用できるはずであるので、その点を考慮すること。

3.3.2 課題

課題 3.1　空スタックを作成してそのポインタ (ダミーセルへのポインタ）を返す関数

```
struct cell *makeNullStack(void);
```

を作成せよ。

【解答例】

```
/*
 *  Make null stack and return pointer to stack
 */
struct cell *makeNullStack(void)
{
  return makeNullList();  /* call makeNullList() */
}
```

課題 3.2　指定されたスタックが空 (1）か否 (0）かを返す関数

```
int emptyStack(struct cell*);
```

を作成せよ。

課題 3.3　引数として指定されたスタックのトップセルが保持しているデータを返す関数

```
int topStack(struct cell*);
```

を作成せよ。

【解答例】

```
/*
 *  Return data of the top stack
 */
int topStack(struct cell *stack)
{
  return retrieveCell(firstCell(stack), stack);
}
```

課題 3.4　引数として指定されたスタックのトップデータを返し、そのセルを取り除く関数

```
int popStack(struct cell*);
```

を作成せよ。

課題 3.5 引数として与えられたデータをスタックに追加する関数

```
void pushStack(int, struct cell*);
```

を作成せよ。

課題 3.6 引数として指定されたスタックに残っているすべてのセルを削除し、スタックのダミーセルまでも削除する関数

```
void deleteStack(struct cell*);
```

を作成せよ。

課題 3.7 ここまでに作成したスタック操作関数群の動作を確認するような `main()` 関数を作成し、動作を確かめよ。

3.4 発展課題: キュー (待ち行列) の実装

3.4.1 実装の仕様

- 内部構造として連結リストを用いる。
- 先週作成したリスト操作ライブラリをできる限り活用する。
- キューの先頭を連結リストの先頭とするか末尾とするかは各自で決めてよい。
- 扱うデータは `int` 型のデータとする。
- 操作関数は、キューの新規作成（MAKENULL）、先頭データの取得（FRONT）、先頭データの取り出し（DEQUEUE）、データの新規追加（ENQUEUE）、空キューの検査（EMPTY）にキュー全体の削除を加えた計 6 関数から構成する。

また、実使用時に利用しやすいように、構造体宣言、関数のプロトタイプ宣言等はヘッダファイル（例えば `queue.h`）に、操作関数群と操作プログラムは別ファイル（例えば `queue.c` と `queue-test.c`）とし、コンパイルには `make` を用いるようにすること。

なお、前回作成したリスト操作ライブラリを利用することから、今回の課題において構造体メンバへの直接アクセスは不要となるはずである。課題回答中にこのような操作が必要となった場合には、リスト操作関数のいずれかで代用できるはずであるので、その点を考慮すること。

課題 3.8 空キューを作成してそのポインタ（ダミーセルへのポインタ）を返す関数

```
struct cell *makeNullQueue(void);
```

を作成せよ。

課題 3.9 指定されたキューが空（1）か否（0）かを返す関数 `emptyQueue` を作成せよ。プロトタイプ宣言は次のとおりとする。

```
int emptyQueue(struct cell*);
```

課題 3.10　引数として指定されたキューの先頭データを返す関数 frontQueue を作成せよ。プロトタイプ宣言は次のとおりとする。

```
int frontQueue(struct cell*);
```

課題 3.11　引数として指定されたキューの先頭データを返し、そのセルを取り除く関数 dequeue を作成せよ。プロトタイプ宣言は次のとおりとする。

```
int dequeue(struct cell*);
```

課題 3.12　引数として与えられたデータをキューに追加する関数 enqueue を作成せよ。プロトタイプ宣言は次のとおりとする。

```
void enqueue(int, struct cell*);
```

課題 3.13　引数として指定されたキューに残っているすべてのセルを削除し、キューのダミーセルまでも削除する関数

```
void deleleteQueue(struct cell*);
```

を作成せよ。

課題 3.14　ここまでに作成したキュー操作関数群の動作を確認するような main() 関数を作成し、動作を確かめよ。

3.5　発展課題: 配列によるスタック・キューの実装

課題 3.15　配列を用いたスタック操作関数群を作成せよ。
各操作関数のプロトタイプは下記の通りとする。

```
struct array_stack *makeNullStack(void);
int emptyStack(struct array_stack*);
int topStack(struct array_stack*);
int popStack(struct array_stack*);
void pushStack(int, struct array_stack*);
void deleteStack(struct array_stack*);
```

また、構造体 struct array_stack は下記を用いることとする。

```
#define MAX_STACK_SIZE 10
struct array_stack {
  int length;                    /* スタックの長さ（格納数）*/
  int elements[MAX_STACK_SIZE];  /* データを格納するための配列 */
};
```

スタックに格納するデータは int 型に限定する。また、格納位置が 0 のものをスタックの底とする。なお、MAX_STACK_SIZE を越えてデータを格納しようとした場合にはエラーメッセージを出し、その操作を無効とすること。

課題 3.16　配列を用いたキュー操作関数群を作成せよ。
各操作関数のプロトタイプは下記の通りとする。

```
struct array_queue *makeNullQueue(void);
int emptyQueue(struct array_queue*);
int frontQueue(struct array_queue*);
int dequeue(struct array_queue*);
void enqueue(int, struct array_queue*);
void deleteQueue(struct array_queue*);
```

また、構造体 `struct array_queue` は下記を用いることとする。

```
#define MAX_QUEUE_SIZE 10
struct array_queue {
  int front;                    /* キューのフロントを示す配列の添字 */
  int length;                   /* キューの長さ（格納数）*/
  int elements[MAX_QUEUE_SIZE]; /* データを格納するための配列 */
};
```

キューに格納するデータは `int` 型に限定する。また、格納位置が配列の終端（`MAX_QUEUE_SIZE-1`）になった場合には、配列の先頭の 0 に折り返すこととし、常に `MAX_QUEUE_SIZE` 個のデータを格納できるようにすること。なお、`MAX_QUEUE_SIZE` を越えてデータを格納しようとした場合にはエラーメッセージを出し、その操作を無効とすること。

3.6 課題: 後置記法の数式の計算

スタックの応用例として、後置記法で書かれた数式を計算するプログラムを作成する。

我々が普段目にする 2 項演算子は、たいてい、2 つの演算数の間に演算子を書く。32 + 9, 4 / 2 といった具合である。これを**中置記法**（infix notation）という。しかし、算術式の書き方はこれだけではない。演算子を 2 つの演算数の前に置く**前置記法**（prefix notation）、2 つの演算数の後ろに置く**後置記法**（postfix notation）（または**逆ポーランド記法**）という書き方もある。ここでは後置記法のみもう少し詳しく説明することにしよう。

中置記法による式 E の後置記法は次のように再帰的に定義される。

1. E が変数または定数であれば、E の後置記法は E である。
2. 任意の 2 項演算子 op について、$E_1 \,\mathrm{op}\, E_2$ の後置記法は $E_1'\ E_2'\ \mathrm{op}$ である。ここで E_1', E_2' はそれぞれ E_1, E_2 の後置記法である。
3. (E) の後置記法は E' である。ここで E' は E の後置記法である。

課題 3.17　中置記法の数式 $32 + 3$, $9 + 4 * 3$, $(1 + 2) * 4$ を後置記法に直せ。

後置記法で書かれた算術式は、スタック 1 つで計算できるという特徴を持つ。式を左から右に 1 トークンずつ読み、次の動作を行えばよい。

1. トークンが数であれば、その数をスタックに積む（push）。
2. トークンが演算子 op であれば、スタックの上 2 つの要素（x, y とする）をおろし（pop）、$x \,\mathrm{op}\, y$ を計算し、結果をスタックに積む。

これを式の終わりまで行い、最後にスタックに残った数が式の計算結果である。

課題 3.18　課題 3.17 で求めた後置記法の数式について、スタックによる計算を手で行ってみよ。スタックの状態の変

化を図示せよ。

課題 3.19　後置記法で書かれた数式を計算するプログラムを作成せよ。

- 入力：後置記法による数式の書かれたファイル 1 つ。
  後置記法の数式は、整数または四則演算子（$+, -, *, /$）が 1 個以上並んだ文字列とする。整数や四則演算子は、1 個の空白または 1 個の改行で区切られているものとする。ただし、どうしてもこの通りに実装できない場合は、必ず 1 個の改行で区切られているとして実装してもよい。
  式の長さや式中に現れる整数には制限を設けない。すなわち、どれだけ式の長さが長くても、またどのような整数が出現しても正しく動作するようにすること。
- 出力：入力の数式を計算した結果。標準出力に出力する。
- 動作に関する仕様：/ の結果は小数点以下切り捨てとする。

プログラムの実行例は次のようになる。以下の例では、作成したプログラム名は a.out であり、カレントディレクトに置かれているとしている。

```
% cat testdata.txt
16 3 + 3 /
-2 -
% ./a.out testdata.txt
8
```

実装に用いるスタックは課題 3.1〜3.6 で実装したものを用いて構わない。さらに、適宜拡張を加えても構わない。拡張を加えた場合は、どのように拡張したかをレポートに明記すること。

3.7　課題：まとめ

課題 3.20　今回実装したプログラムについて、改良する点があるとすればどのようなことが考えられるか。また、その改良を実装する場合の実装方針、問題点、難しいと思われる点などについて考察せよ。

第 4 章

再帰

4.1 定義

関数 f が、その関数本体の中に f 自身の適用を含むとき、関数 f を再帰関数 (recursive function) という。一般的には、f から直接的に f が呼び出される場合だけではなく、間接的に呼び出される場合も再帰関数という。

特に、関数 f の中で直接的にたかだか 1 回しか f を呼び出さない場合を線形再帰 (linear recursive)、2 回以上の f の呼び出しを含む場合を非線形再帰 (non-linear recursive) という。

再帰関数には、必ず以下の 2 つの部分が含まれる。

1. 基底 (basis)…再帰を含まない計算
2. 帰納段階 (induction step)…再帰を含む計算

再帰関数は、リストや木などの再帰的なデータ構造[*1]を扱うアルゴリズムにおいて特に強力な計算手段であることが知られている。

4.1.1 再帰関数の例

ここではプログラミング言語 ML を用いて再帰関数の例を示す。

■階乗計算　自然数 n の階乗は以下のアルゴリズムで計算できる。

- $n = 0$ のとき 1
- $n \geq 1$ のとき $n * (n - 1)!$

これを ML で書くと次のようになる。

```
fun factorial(n) =
    if n = 0 then 1 else n * factorial(n - 1);
```

■リストの逆順並び替え　リスト L を逆順に並び替える操作は以下のアルゴリズムで計算できる。

- L が空リストのとき、結果は空リストである。
- L が空リストでないとき、$tl(L)$ を逆順に並び替えたリストの後ろに、$hd(L)$ だけからなるリストを連接したも

[*1] フラクタルなども再帰的なデータ構造を持っている。

のになる。ただし、$hd(L)$, $tl(L)$ はそれぞれ L の頭部、尾部を表す。

これを ML で実装すると次のようになる。

```
fun reverse(L) =
    if L = nil then nil
               else reverse(tl(L)) @ [hd(L)];
```

4.1.2 再帰関数の実装上の注意

再帰関数を実装する場合、いくつか決まりごとがある。

- 基底、帰納段階を必ず両方とも作り、引数の値などによって条件分岐させる。たとえば整数を引数に取る関数の場合は 0(もしくは 1) かどうかで、またリストを引数に取る場合は、リストが nil かどうかで、基底・帰納段階のいずれかに条件分岐させることがひじょうに多い。
- 帰納段階での再帰呼び出しでは、必ず引数の値を小さくして呼び出すようにする。たとえば、整数の場合は $n-1$、リストの場合は $tl(L)$ などとする。これと、上記の条件分岐により、再帰関数はいつか必ず基底の計算を行い (無限に帰納段階を繰り返すことはない)、計算が停止することが保証される。

4.2 再帰プログラミング

4.2.1 数に関する再帰

例題 4.1　1 から引数で指定された正の整数値までの和を返す関数 `int sigma(int);` を再帰を用いて作成せよ。

$$
\begin{aligned}
sigma(n) &= n + sigma(n-1) \\
&= n + (n-1) + sigma(n-2) \\
&\vdots \\
&= n + (n-1) + \cdots + sigma(1) \\
&= n + (n-1) + \cdots + 1
\end{aligned}
$$

【解答例】

```
int sigma(int num)
{
  int result;  /* calculation result */

  if (num <= 1) {
    result = num;
  } else {
    result = num + sigma(num - 1);
  }

  return result;
}
```

【main 関数の例】

```
int main(int argc, char *argv[])
{
  /* check arguments */
  if (argc != 2) {
    fprintf(stderr, "Usage: %s num\n", argv[0]);
    exit(1);
  }

  /* check argument sign */
  if (atoi(argv[1]) < 0) {
    fprintf(stderr, "Error: The argument must be a positive value.\n");
    exit(1);
  }

  /* calculation and print result */
  printf("sigma(%d) = %d\n", atoi(argv[1]), sigma(atoi(argv[1])));

  return 0;
}
```

課題 4.1 (発展) 上記の解答例では、引数として与える値に制限が存在する。その制限値の上限についてあらかじめ推定せよ。また、様々な値を引数として与え、その推定の正否を確認するとともに、その様な制限が発生している理由を文献等により調査せよ。

課題 4.2 $1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$ という数列を**フィボナッチ数列** という。この数列の最初の 2 要素は 1 であり、それ以降の要素はその前 2 つの要素の和となっている。例えば 5 番目の要素は、その前 2 つの要素である 3, 4 番目の要素 2 と 3 の和であるから $2 + 3 = 5$ となっている。

フィボナッチ数列において、引数で指定された順番に位置する要素を返す関数 `int fibonacci(int);` を再帰を用いて作成せよ。また、この関数の動作を確かめる `main()` 関数を作成し、動作を確かめよ。

$$\begin{aligned} fibonacci(n) &= fibonacci(n-1) + fibonacci(n-2) \\ &= \{fibonacci(n-2) + fibonacci(n-3)\} + \{fibonacci(n-3) + fibonacci(n-4)\} \\ &\vdots \end{aligned}$$

4.2.2 リストに関する再帰

これ以降の課題については、前回に作成したリスト操作関数を用いること。

課題 4.3 リストへのポインタを引数としてとり、その内容を逆順に並べ替えたリストへのポインタを返す関数 `struct cell *reverseList(struct cell*);` を再帰を用いて作成せよ。また、この関数の動作を確かめる `main()` 関数を作成し、動作を確かめよ。なお、並べ替え後に元のリストが破壊されていても構わない[*2]。

$$\begin{aligned} reverseList([3,6,4,5,2]) &= reverseList([6,4,5,2])@[3] (@\text{はリストの連結演算を表す}) \\ &= reverseList([4,5,2])@[6]@[3] \\ &\vdots \\ &= [2,5,4,6,3] \end{aligned}$$

[*2] すなわち、`printList(list)` の結果が変化しても構わない。

課題 4.4　与えられたリスト `list` を二分割した結果を `list1`、`list2` として返す再帰関数を作成せよ。プロトタイプ宣言は次のとおりとする。

```
void divideList(struct cell *list, struct cell *list1, struct cell *list2);
```

分割は、`list` の奇数番目のデータを `list1` に、偶数番目のデータを `list2` に格納するという手順で行うこと。なお、並べ替え後に元のリストが破壊されていても構わない[*3]。

また、この関数の動作を確かめる `main()` 関数を作成し、動作を確かめよ。

参考) $pcList = [1, 2, 3, 4, 5]$ のとき、

$$\begin{cases} list1 &= [1]@([3,4,5] \text{ を divideList した結果得られる } list1) \\ &= [1]@([3]@([5] \text{ を divideList した結果得られる } list1))、 \\ list2 &= [2]@([3,4,5] \text{ を divideList した結果得られる } list2) \\ &= [2]@([4]@([5] \text{ を divideList した結果得られる } list2)) \end{cases}$$

課題 4.5　与えられた 2 つのリストを結合する再帰関数を作成せよ。プロトタイプ宣言は次のとおりとする。

```
void mergeList(struct cell *list1, struct cell *list2, struct cell *merged);
```

結合は、`list1` および `list2` それぞれの先頭のデータを比較し、小さい方のデータを `merged` に追加するという手順で行う。なお、並べ替え後に元のリスト `list1` および `list2` が破壊されていても構わない[*4]。

また、この関数の動作を確かめる `main()` 関数を作成し、動作を確かめよ。

参考) $list1 = [3, 6, 2]$、$list2 = [4, 5, 7]$ のとき、

$$\begin{aligned} merged &= [3]@([6,2]、[4,5,7] \text{ を mergeList した結果得られる merged}) \\ &= [3]@([4]@([6,2]、[5,7] \text{ を mergeList した結果得られる merged})) \\ &= \cdots \end{aligned}$$

課題 4.6　作成した `divideList` と `mergeList` を利用することで、リストの並べ替えを行うことができる。並べ替えを行う関数が `mergeSort` であるとき、並べ替えは、

1. 与えられたリストを `divideList` によって二分割、
2. 分割した結果をそれぞれ `mergeSort` で並べ替え、
3. 並べ替えた結果 2 つを `mergeList` で結合、

という手順で実現できる。この手法は **マージソート** として知られている。

与えられたリスト `list` からその内容を昇順に並べ替えたリスト `sorted` を得る再帰関数を作成せよ。プロトタイプ宣言は以下のようになる。

```
void mergeSort(struct cell *list, struct cell *sorted);
```

また、この関数の動作を確かめる `main()` 関数を作成し、動作を確かめよ。

参考) `list = [ 4, 7, 2, 3, 9, 1 ]` が与えられたとき、`mergeSort` を実行した結果得られる `sorted` は `sorted = [ 1, 2, 3, 4, 7, 9 ]` となる。

$$[4,7,2,3,9,1] \rightarrow \boxed{\texttt{divideList}} \rightarrow \left\{ \begin{array}{l} [4,2,9] \rightarrow \boxed{\texttt{mergeSort}} \rightarrow [2,4,9] \\ [7,3,1] \rightarrow \boxed{\texttt{mergeSort}} \rightarrow [1,3,7] \end{array} \right\} \rightarrow \boxed{\texttt{mergeList}} \rightarrow [1,2,3,4,7,9]$$

[*3] すなわち、`printList(list)` の結果が変化しても構わない。

[*4] すなわち、`printList(list1)`、`printList(list2)` の結果が変化しても構わない。

さらに $[4,2,9]$ の `mergeSort` は次のようになる。

$$[4,2,9] \rightarrow \boxed{\texttt{divideList}} \rightarrow \left\{ \begin{array}{c} [4,9] \rightarrow \boxed{\texttt{mergeSort}} \rightarrow [4,9] \\ [2] \rightarrow \boxed{\texttt{mergeSort}} \rightarrow [2] \end{array} \right\} \rightarrow \boxed{\texttt{mergeList}} \rightarrow [2,4,9]$$

4.3 課題：まとめ

課題 4.7 今回実装したプログラムについて、改良する点があるとすればどのようなことが考えられるか。また、その改良を実装する場合の実装方針、問題点、難しいと思われる点などについて考察せよ。

第 5 章

木

木は、集合の要素に階層構造を与えるときにしばしば用いられる概念であり、情報科学の分野でもっとも重要なデータ構造の一つである。木の定義、基本的な用語などについては添付の資料を参照のこと。

木のある節点に着目すると、その子節点を根とする木 (部分木) が必ず存在する。この意味で、木は再帰的なデータ構造であり、木を扱うアルゴリズムは再帰で考えると簡単になるものが多い。とくに典型的なものが、木の節点をすべてたどるアルゴリズムのうち**深さ優先探索** *(depth first search)* と呼ばれるものである。このバリエーションとして行きがけ順 (preorder)、通りがけ順 (inorder)、帰りがけ順 (postorder) という 3 種類の全節点の順序づけの方法が知られている。

5.1 2 分木

今回の実験では、簡単のため、*2* **分木**と呼ばれる特殊な木のみ扱うことにする。2 分木は、節点の子がたかだか 2 つしかない (つまり、0、1、もしくは 2) 木である。左側の子節点を根とする木を左部分木、右側の子節点を根とする木を右部分木という。2 分木の基本操作群の一例を図 5.1 に示す。

$Node\ CREATENODE(Label\ label, Node\ left, Node\ right)$　$label$ をラベルとする節点を根、$left$ を根とする木を左部分木、$right$ を根とする木を右部分木とする 2 分木を生成する。
$Node\ LEFTNODE(Node\ n)$　節点 n の左部分木の根である節点を返す。
$Node\ RIGHTNODE(Node\ n)$　節点 n の右部分木の根である節点を返す。

図 5.1　2 分木の基本操作群

2 分木の節点は、子節点がたかだか 2 つと限られているので、次のような属性から構成される構造体として実現すると簡単である。

- その節点につけられているラベル
- 左部分木の根である節点へのポインタ。左部分木がない場合は NULL。
- 右部分木の根である節点へのポインタ。右部分木がない場合は NULL。

5.2 二分木

次回の実験で利用しやすいように、構造体宣言、関数のプロトタイプ宣言等はヘッダファイルに、操作関数群と操作プログラムは別ファイルとし、コンパイルには make を用いるようにすること。

課題 5.1 二分木の節点を表す構造体 struct node を作成せよ。なお、保持するデータは int 型のデータ 1 つのみとする。

課題 5.2 引数として、格納データおよび左の子へのポインタ、右の子へのポインタをこの順にとり、その内容の節点を新規に作成してそのポインタを返す関数

```
struct node *createNode(int, struct node*, struct node*);
```

を作成せよ。一方あるいは双方の子が存在しない節点を作成する場合には、子として NULL を指定することとする。また、節点の格納領域は malloc を用いて動的に確保すること。

課題 5.3 引数として節点へのポイントをとり、その節点の左の子/右の子の節点へのポインタを返す関数

```
struct node *leftNode(struct node*);
struct node *rightNode(struct node*);
```

を作成せよ。なお、該当する子が存在しない場合には NULL を返すこと。

課題 5.4 引数として節点へのポインタをとり、その節点が保持しているデータを返す関数

```
int labelNode(struct node*);
```

を作成せよ。

課題 5.5 引数として節点へのポインタをとり、その節点を根とする二分木を深さ優先でたどり、各節点が保持しているデータを 行きがけ順に 表示する再帰関数

```
void preOrder(struct node*);
```

を作成せよ。

課題 5.6 引数として節点へのポインタをとり、その節点を根とする二分木を深さ優先でたどり、各節点が保持しているデータを 通りがけ順に 表示する再帰関数

```
void inOrder(struct node*);
```

を作成せよ。

課題 5.7 引数として節点へのポインタをとり、その節点を根とする二分木を深さ優先でたどり、各節点が保持しているデータを 帰りがけ順に 表示する再帰関数

```
void postOrder(struct node*);
```

を作成せよ。

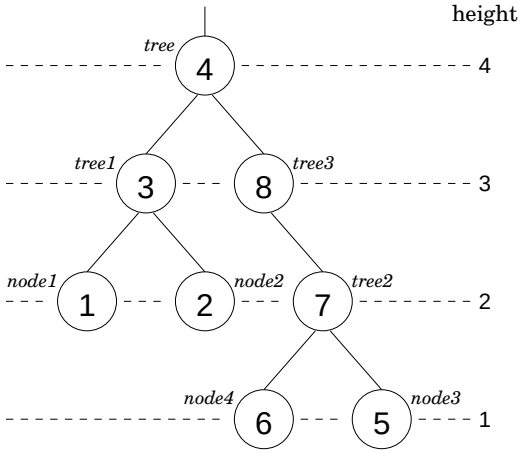
課題 5.8 引数として節点へのポインタをとり、その節点を根とする二分木（部分木）の高さを返す再帰関数

```
int heightTree(struct node*);
```

を作成せよ。

参考) 二分木の高さは、左右それぞれの子を根とする二分木（部分木）のうち高い方の高さ ＋1 で求めることができる。

```
heigthTree(node) = MAX( heigthTree(node の左の子), heigthTree(node の右の子)) + 1;
```



課題 5.9　引数としてあるデータと節点へのポインタをとり、その節点を根とする二分木（部分木）中に存在する、そのデータと同じ値をもつ節点へのポインタを返す再帰関数

```
struct node *locateNode(int, struct node*);
```

を作成せよ。ただし、該当する節点がない場合には NULL を返すこと。なお、該当する節点が複数存在する場合であっても最初に見つかったもののみを返せばよい。

課題 5.10　引数として節点へのポインタをとり、その節点を根とする二分木（部分木）を削除する再帰関数

```
void deleteTree(struct node*);
```

を作成せよ。

参考) 二分木の削除は、その二分木を構成する全ての節点を再帰的に削除することで実現できるが、削除する順番に注意すること。

課題 5.11　実装した二分木の操作関数が正しく動作するか確かめるように main() 関数を作成し、正しく動作していることを確かめよ。

5.3 課題：まとめ

課題 5.12　今回実装したプログラムについて、改良する点があるとすればどのようなことが考えられるか。また、その改良を実装する場合の実装方針、問題点、難しいと思われる点などについて考察せよ。

第 6 章

2 分探索木

計算機で大量のデータを扱う場合、単なる配列や連結リストでは検索・挿入・削除などの操作がひじょうに遅くなる。たとえば配列で 10,000 件のデータを持っているときに、その先頭に 1 つ要素を挿入しようとすると、10,000 件のデータをすべて 1 つずつ後ろへずらさなければならない[*1]。そこで、より効率の良いデータ操作が行えるデータ構造がいろいろ考えられている。

この実験では、それらの中で比較的簡単な 2 分探索木 (binary search tree) を取り上げる。これは 2 分木の特殊な場合であり、先に述べた 2 分木を用いて実装することができる。

6.1 定義

2 分探索木は、次の規則に基づいて各節点に集合の要素を 1 つずつ対応づけた 2 分木である[*2]

- 任意の節点 x に対し、x の左部分木に格納された要素はすべて x より小さい。
- 任意の節点 x に対し、x の右部分木に格納された要素はすべて x より大きい。

図 6.1 に 2 分探索木の例を示す。

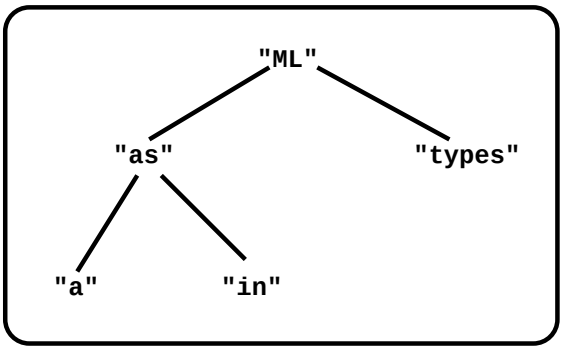


図 6.1　2 分探索木の例

[*1] アルゴリズムの計算量が $O(n)$ (データ数 n に比例する) であるという。これは、検索ではかなり時間がかかる部類に入る。

[*2] ここでは「小さい」「大きい」という言葉を用いて直感的に説明しているが、厳密には全順序 (total order) であればよい。整数や実数の大小関係、文字列の辞書順などは全順序の例である。

6.2　基本操作群

2 分探索木に対する基本操作群には図 6.2 のようなものが考えられる。これらの操作はいずれも再帰的な手続きに

$MEMBER(x, A)$　要素 x が 2 分探索木 A に含まれているかどうかを調べる。
$INSERT(x, A)$　要素 x を 2 分探索木 A に挿入する。結果は 2 分探索木になっていなければならない。
$MIN(A)$　2 分探索木 A のうち、最小の要素に対応づけられている節点を得る。
$DELETE(x, A)$　要素 x を 2 分探索木 A から削除する。結果は 2 分探索木になっていなければならない。

図 6.2　2 分探索木に対する基本操作群

よって記述することができる。たとえば $MEMBER(x, A)$ は次のようになる。

```
boolean MEMBER(x, A) {
    if (A == nil) {
        return false; /* 2 分探索木は空 */
    } else if (x == A.iData) {
        return true;  /* A の根と x が一致 */
    } else if (x < A.iData) {
        return MEMBER(x, A.leftTree);  /* x があるなら左部分木の中 */
    } else {  /* x > A.iData */
        return MEMBER(x, A.rightTree); /* x があるなら右部分木の中 */
    }
}
```

$INSERT(x, A)$, $DELETE(x, A)$ は、結果がまた 2 分探索木になっていなければならないため、実装が複雑になる。例として $DELETE(x, A)$ のアルゴリズムの概略を示す。

1. A が空であれば (A には x は含まれていなかったので) 何もしない。
2. A が空でなければ
   (a) $x < A.iData$ であれば、x が含まれているとすれば左部分木の中なので、A の左部分木から x を消去し、結果として得られた 2 分探索木を A の左部分木とする。
   (b) $x > A.iData$ であれば、x が含まれているとすれば右部分木の中なので、A の右部分木から x を消去し、結果として得られた 2 分探索木を A の右部分木とする。
   (c) $x == A.iData$ であれば、A の根を消すことになる。
      i. A が葉 (左部分木も右部分木もない) であれば、単に A を消す。
      ii. A に右部分木がなければ、A の根を削除して得られる 2 分探索木は A の左部分木そのものになる。
      iii. A に左部分木がなければ、A の根を削除して得られる 2 分探索木は A の右部分木そのものになる。
      iv. A に左部分木も右部分木もあれば、右部分木中の最小の要素を求める。これを y とすると、x を削除して得られる 2 分探索木は、根が y、左部分木が $A.leftTree$、右部分木が $DELETE(y, A.rigthTree)$ となる。

6.3 二分探索木

6.3.1 実装上の注意

二分探索木はその内部構造が二分木であるため、前回作成した二分木の操作関数群をそのまま使用することとする。

また、二分木を内部構造とすることから分かるように、課題のほとんどは再帰処理によって実現されるものであることに注意すること。

なお、本課題中の解答例として、作成する関数の雛形を示しているが、この中の日本語のコメントの部分は各自でそれに相当する処理を記述する必要がある部分である。

6.3.2 課題

課題 6.1 引数として参照データと二分探索木の根へのポインタをとり、参照データが二分探索木に含まれているかどうかを返す関数

```
int BS_member(int, struct node*);
```

を作成せよ。返り値は、参照データが含まれている場合には `1`、含まれていない場合には `0` を返すこととする。

【解答例】

```
/***
 ***  test that a node that has specified label is in the tree
 ***/
int BS_member(int data, struct node *tree)
{
  int ret = 0;  /* result (0: not member, 1: member) */

  if (tree != (struct node*)NULL) {
    /* 節点 tree が存在する場合の処理
       存在しない場合にはデータ data も存在しない */
  }

  return ret;
}
```

課題 6.2　引数として追加するデータと二分探索木の根へのポインタをとり、追加後の二分探索木の根へのポインタを返す関数

```
struct node *BS_insert(int, struct node*);
```

を作成せよ。なお、新規の二分探索木作成にもこの関数を使用するため、その点も考慮すること[*3]。

【解答例】

```
/***
 ***  insert a node that has specified label
 ***/
struct node *BS_insert(int data, struct node *tree)
{
  if (tree == (struct node*)NULL) {
    /* 節点 tree が存在しない場合（新規に二分探索木を作成する場合）の処理 */
  } else {
    /* 節点 tree が存在する場合の処理 */
  }

  return tree;
}
```

課題 6.3　引数として二分探索木の根へのポインタをとり、最小のデータを保持する節点へのポインタを返す関数

```
struct node *BS_min(struct node*);
```

を作成せよ。

【解答例】

```
/***
 ***  get a node that has minimum label in the tree
 ***/
struct node *BS_min(struct node *tree)
{
  struct node *min = (struct node*)NULL;  /* pointer to minimum node (initially NULL) */

  if (tree != (struct node*)NULL) {  /* check tree is exist or not */
    if (leftNode(tree) == (struct node*)NULL) {
      /* 節点 tree が左部分木を持っていない場合の処理 */
    } else {
      /* 節点 tree が左部分木を持っている場合の処理 */
    }
  }

  return min;
}
```

課題 6.4　引数として削除データの値と二分探索木の根へのポインタをとり、削除後の二分探索木の根へのポインタを返す関数

```
struct node *BS_delete(int, struct node*);
```

を作成せよ。

[*3] 具体的には、根へのポインタが NULL の場合には二分探索木の新規作成と見なして処理を行うようにする。

【解答例】

```
/***
 ***  delete a node that has a label equals to specified label
 ***/
struct node *BS_delete(int data, struct node *tree)
{
  int tmp;  /* temporary data */

  if (tree != (struct node*)NULL) {  /* check tree is exist or not */
    if (labelNode(tree) > data) {
      /* データ data が節点 tree のデータより小さい場合の処理 */
    } else if (labelNode(tree) < data) {
      /* データ data が節点 tree のデータより大きい場合の処理 */
    } else {
      /* data is equal to this node, then delete this node */
      if ((leftNode(tree) == (struct node*)NULL)
          && (rightNode(tree) == (struct node*)NULL)) {
        /* 節点 tree が葉である場合の処理 */
      } else if (leftNode(tree) == (struct node*)NULL) {
        /* 節点 tree が右部分木しか持っていない場合の処理 */
      } else if (rightNode(tree) == (struct node*)NULL) {
        /* 節点 tree が左部分木しか持っていない場合の処理 */
      } else {
        /* 節点 tree が左右の部分木を持っている場合の処理 */
      }
    }
  }

  return tree;
}
```

課題 6.5　引数として二分探索木の根へのポインタをとり、格納しているデータ全てを昇順に表示する関数

```
void BS_print(struct node*);
```

を作成せよ。

課題 6.6　引数として二分探索木の根へのポインタをとり、その二分探索木の全ての節点を削除する関数

```
void BS_destroy(struct node*);
```

を作成せよ。

課題 6.7　以上の課題で実装した関数の動作を確かめるような `main()` 関数を実装し、動作を確かめよ。

6.4　課題：まとめ

課題 6.8　今回実装したプログラムについて、改良する点があるとすればどのようなことが考えられるか。また、その改良を実装する場合の実装方針、問題点、難しいと思われる点などについて考察せよ。

以下の課題は成績評価には加えませんので、なるべく正直に書いてもらえると助かります。

課題 6.9　今回の 6 週の実験を通して自分が学んだ (り学 { ば | べ } なかった) ことを自分なりに考察し、今後のことも含めて書いて下さい。6 週の実験を通しての感想なども適宜織り交ぜていただいて構いません。