演習解答例

１－１

（１） 効率＝0 は出力＝0 のとき。例えば、一生懸命作ったプログラムが全く利用されないとき。または、プログラム商品で全く売れないとき。（注：あまり利用されないプログラムは、エラー発生の報告があまりないため、信頼性が高くなる）

（２） 効率＝∝は入力＝0 のとき。例えば、ある目的で作ったプログラムがまったく別の目的に利用できる場合、入力(作成費用)0 で出力(プログラム)が得られる。（注：もっとも効率の良い作り方は、作らないで、あるものを利用すること）

１－２

例えば、家の夕食の料理を考えると、

- 分析：家族の健康状態、お父さんの帰宅時間、財布の中身と給料日までの日数から可能な出費の割り出し、その日の気候、スーパーの特売品広告などを分析する。
- 設計：分析結果をもとに夕食の献立を決める。材料、それぞれの料理の手順、盛り付けなどを設計する。
- 実装：冷蔵庫の中身を見て、必要な材料の買出しから、実際の調理、盛り付けまで、夕食を実装する。
- テスト：だしの加減をみたり、できた料理を少し試食し、美味しくできているかどうかをテストする。

問題指向／解法指向：

- 問題指向：美味しさ、栄養、コストなどを重視
- 解法指向：調理法、台所の設備、調理の手間などを重視

人間指向／コンピュータ指向

- 人間(料理をする人)指向：料理をする人が考える作業。例えば盛り付け方など。
- コンピュータ(料理される食材や道具)指向：料理される食材や、鍋釜・レンジなどの道具の状態、温度、焼き加減などを調整する作業。

１－３　例えば、

- ウォータフォール型：大切なお客様を招待する食事。お客様の好みを調査分析し、十分に計画を練りながら準備をすすめ、料理を行う。
- プロトタイピング型：子供のお誕生日会の食事。とりあえず自分の子供を相手に当日の料理のサンプルを作り、評価する(プロトタイピング)。その後、そのサンプルを参考にして本番の料理にもっていく。
- スパイラル型：新婚の妻が夫の両親を招待する食事。毎日の料理を夫に食べさせながら、徐々に夫の家の味にしていく。そして、夫の両親にその味を振舞う。
- ＸＰ型：冷蔵庫の残り物を使った食事。妻と子供は出前寿司(松)で夕食。その後、残業で遅くなって帰ってきた夫に冷蔵庫の残りもので適当に料理して出す。「これ、う

まいね」と喜ぶ夫。そこでその料理の仕方を記録し、新たなレパートリーに加える。で、結局ＸＰ型が（好むと好まざるに関わらず）一番多いかも？

１－４　　図１．８参照

命令型：

- 手続き型：データに対する命令(読み書き)の実行順序を規定した処理のまとまり（手続き）を基準にプログラムを作成する方式。
- オブジェクト指向型：命令によって処理されるデータや装置（オブジェクトと呼ぶ）とそれに対する操作のまとまりを基準にプログラムを作成する方式。

宣言型：

- 関数型：数学的な意味での関数（パラメータの値だけで結果が定まり、内部状態を持たない）の合成による計算の規定によって、プログラムを作成する方式。
- 論理型：述語論理式（「○○は××の合計である」などの表現）の組合せで、述語の意味（仕様）を記述することによって、プログラムを作成する方式。

２－１

もとのスパゲッティプログラム

```
01      P1;
02  L1:P2;
03  L2:if(C1) goto L4;
04      P3;
05      P4;
06      if(C2) goto L3;
07      P5;
08      P6;
09  L3:if(C3) goto L2;
10      P7;
11  L4:if(C4) goto L5;
12      P8;
13      goto L1;
14  L5:
```

(1)02 の L1:と 13 の goto L1 を while ループに変更して L1 を消去
　(goto L5 を break に変換)
(2)06 から 09 の文を if 構造に変更して L3 を消去

```
01    P1;
02    while(true) { /*(1)*/
03        P2;
04     L2:if(C1) goto L4;
05        P3;
06        P4;
07        if(!C2) { /*(2)*/
08            P5;
09            P6;
10        } /*(2)*/
11        if(C3) goto L2;
12        P7;
13     L4:if(C4) break;
14        P8;
15    } /*(1)*/
```

(3)04 から 13 の文を if 構造に変更して L4 を消去

```
01    P1;
02    while(true) {
03        P2;
04     L2:if(!C1) { /*(3)*/
05            P3;
06            P4;
07            if(!C2) {
08                P5;
09                P6;
10            }
11            if(C3) goto L2;
12            P7;
13        } /*(3)*/
14        if(C4) break;
15        P8;
16    }
```

(4)04 から 11 の L2 の繰返しについては、11 からの判断で、

①C3 で!C1 のとき、05 へ移る。

②C3 で C1 のとき、14 へ移る。

③!C3 のとき、12 へ移る。

上で②と③の順序を変えると、

①C3 で!C1 の間、05 からの繰返しを続ける。

②①の繰り返しを出たところで、!C3 なら 12(P7;)へ、

③①の繰り返しを出たところで、C3 なら 13 へ移る。

と変換できる。

(5)さらに、この②と③は if 構造に変換できる。

```
01    P1;
02    while(true) {
03        P2;
04        if(!C1) {
05            do { /*(4)*/
06                P3;
07                P4;
08                if(!C2) {
09                    P5;
10                    P6;
11                }
12            while(C3 && !C1); /*(4)*/
13            if(!C3) { /*(5)*/
14                P7;
15            } /*(5)*/
16        }
17        if(C4) break;
18        P8;
19    }
```

(6)17 の break を制御変数 end の導入で消去する。

```
01    P1; end=false;
02    while(!end) {
03        P2;
```

```
04        if(!C1) {
05            do { /*(4)*/
06                P3;
07                P4;
08                if(!C2) {
09                    P5;
10                    P6;
11                }
12            while(C3 && !C1); /*(4)*/
13            if(!C3) { /*(5)*/
14                P7;
15            } /*(5)*/
16        }
17        if(C4) end=true;
18        else {
19            P8;
20        }
21    }
```

以上

２－２

- 処理のコピーによる変換：
  - 利点：実行効率(処理スピード)は悪化しない。
  - 欠点：コードが大きくなる。また、条件式のコピーでは、条件式が状態を変更しない場合のみＯＫ。
- 制御変数の追加による変換：
  - 利点：コードがあまり大きくならないですむ。
  - 欠点：余分な判定で実行効率が下がる恐れがある。

２－３

図２．８の構造化原理のフローチャートでは、多分岐選択文を繰り返す構造になっており､繰返しは１つだけ現われる。

２－４　例えば、

```
起床
if( 体調 OK ) {
    洗顔と朝食
    通勤
    午前の仕事
    昼食
    午後の仕事
    if( デートの日 ) {
        デートを楽しむ
        帰宅
    } else {
        if( 仕事がかたずかない )
            残業
        仕事のまとめ
        帰宅
        夕食と風呂
    }
} else {
    休暇をとる
}
就寝
```

洗顔と朝食の詳細化：

```
顔を洗う
朝食準備
朝食を取る
歯を磨く
トイレに入る
if( 時間がある ) {
    シャワーを浴びる
    入念に化粧する
} else {
    シャンプーを軽く済ませる
    クリームと口紅だけつける
}
通勤服を着る
```

２－５

- 各命令記述が独立になっていない場合：

  入力処理；

  計算処理；

  出力処理；

  このままではそれぞれ独立に詳細化できない。

  （扱うデータとそれの計算方式が前もって示してあればこれでも OK）

- 命令記述の詳細を見ないと意味が分からない場合

  前処理

  主処理

  後処理

  これでは、ほとんど意味が分からない。

- 不要な詳細まで表現されてしまっている場合

  テーブルポインターを NULL に設定する

  テーブル読込み処理

  まだ、テーブルの実現にポインターを使うかどうかわからない。

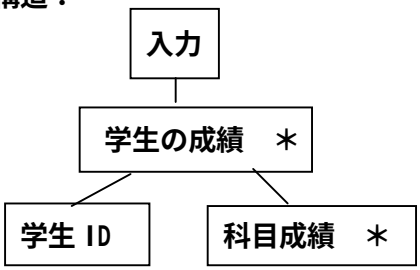
- 詳細さのレベルが不統一な場合

  上の例。

  １行目の表現を「テーブル読込み処理」のレベルに合わせると、「テーブル初期化」とか「テーブルクリア」など。

２－６

入力データ構造：



① 入力データ構造を反映させたプログラム例

[2_6_1.c 参照]

② 入力データ構造を無視したプログラム例

[2_6_2.c 参照]

③

ジャクソン法を使った場合の利点・欠点

- 利点：まとまった処理(例えば学生毎の処理)の順序がプログラム上でも上から下へと記述され、分りやすい。
- 欠点：ファイル入力処理が複数回出現し、ネストが深いプログラムの場合はプログラムサイズが増えることもある。

ジャクソン法を使わなかった場合：上記の反対。

２－７

コメント取出しの状態図

[演習解答図２－７参照]

プログラム例

[2_7.c 参照]

３－１

[3_1.c 参照]

３－２　例えば各ノードには以下の情報をもたせる。

１　葉か、そうでないか。

２　葉でなければ、制御のタイプ（順次、選択、繰返し）
　　制御情報（選択条件、繰返し条件等）
　　子ノードのリスト（順次は２以上、選択は２、繰返しは１）

３　葉なら、　単文か null 文、単文なら単文の情報をもつ

例えば、図２．５のプログラムの木構造を以下に示す。、

```
プログラム＝
  順次
    ＋―単文：Ｐ１
    ＋―単文：end=false
    ＋―繰返しー条件：[前](end==false)
          ＋―順次
                ＋―単文：Ｐ２
                ＋―選択―条件：(！Ｃ１)
                ｜      ＋[Ｔ]―順次
                ｜      ｜          ＋―繰返しー条件：[後]（Ｃ３＆＆！Ｃ１）
                ｜      ｜          ｜      ＋－順次
```

```
|    |        |          +―単文：Ｐ３
|    |        |          +―単文：Ｐ４
|    |        |          +―選択―条件：(！Ｃ２)
|    |        |               +[Ｔ]―順次
|    |        |               |        +―単文：Ｐ５
|    |        |               |        +―単文：Ｐ６
|    |        |               +[Ｆ]―(null)
|    |       +―  選択―条件：(Ｃ３)
|    |             +[Ｔ]―単文：Ｐ７
|    |             +[Ｆ]―(null)
|   +[Ｆ]―(null)
+―  選択―条件：(Ｃ４)
      +[Ｔ]―単文：end=true
      +[Ｆ]―単文：Ｐ８
```

３－３

状態遷移図では、遷移の方向にしか参照しないので、ノードにはそこから出るアーク（始アーク）のみの情報でよい。また、アークにはアークが入るノード（終ノード）だけを持てばよい。従って、ノードとアークは以下の情報を含む。

ノード：　ノード情報：(ノード名、ノード処理)
　　アークリスト(最初のアークへのポインター)
アーク：　アーク情報：(イベント、条件、アーク処理)
　　入るノード
　　次のアーク

そして、特別なアーク（開始アーク）を定める。

図２．２０に対しては、Ｎｉをノード、Ａｊをアークとして、以下ようになる。

[N1:アイドル, ArcList={A1,A2}],
[N2:お金不足/金額表示, ArcList={A3,A4,A5}]
[N3:お金あり/金額＆製品表示, ArcList={A6,A7,A8,A9,A10}]
[N4:返却中, ArcList={A11}]

[A0:開始/金額=0, N=N1]
[A1:入金[入金＜値段]/金額+=入金額, N=N2, nextArc=A2]
[A2:入金[入金≧値段]/金額+=入金額, N=N3, nextArc=NULL]

[A3:入金[(金額＋入金)＜値段]/金額+=入金額，N=N2，nextArc=A4]
[A4:入金[(金額＋入金) ≧値段]/金額+=入金額，N=N3，nextArc=A5]
[A5:返却ボタン,タイムアウト/返金開始，N=N4，nextArc=NULL]
[A6:入金/金額+=入金額，N=N3，nextArc=A7]
[A7:製品ボタン[(金額－値段)≧値段]/製品を出す;金額－=値段，N=N3，nextArc=A8]
[A8:製品ボタン[金額＝値段]/製品を出す;金額－=値段，N=N1，nextArc=A9]
[A9:返却ボタン,タイムアウト/返金開始，N=N4，nextArc=A10]
[A10:製品ボタン[(金額－値段)＜値段]/製品を出す;金額－=値段，N=N2，nextArc=NUL]
[A11:返金終了/金額＝0，N=N1，nextArc=NULL]

この例では、ある状態で受付けられないイベントがくると処理を終わる。
（そのようなイベントは無視するようにプログラムを直すこともできる）

状態遷移の実行プログラムの例：

```
Arc a=A0; // 開始アーク
while( a != NULL ) {
    Node n＝a の遷移先ノード;
    a の処理(アークの処理)を実行;
    n の処理(ノードの処理)を実行;
    Event E = イベント入力;  // イベント待ち
    a = NULL;
    forall Arc aa in n の ArcList  {
        if( aa のイベント==E && aa の条件が真 ) {
            a = aa;
            break;
        }
    }
}
```

３－４　３－３のノード情報に入れ子の状態遷移の情報を追加する。入れ子の状態遷移の情報としては、状態遷移の開始アークをもたせる。すなわち、

ノード：　ノード情報：(ノード名、ノード処理)
　　アークリスト(最初のアークへのポインター)
　　入れ子の開始アーク（なければ NULL）

とする。

状態遷移プログラム例（第 5 章で述べる再帰関数を利用する）。

```
/* 対応するイベントが見つかる限り遷移を実行。対応がなかったイベントを返す */
Event DoSTD(Arc arc) {
    Event E; Arc a, a_son, aa; Node n;
    a = arc;
    while( a != NULL ) {
        n＝a の遷移先ノード;
        a の処理(アークの処理)を実行;
        n の処理(ノードの処理)を実行;
        a_son = n の入れ子の開始アーク
        If( a_son != NULL )
            E = DoSTD(a_son);    // 入れ子状態遷移の実行
        else
            E = イベント入力();  // イベント待ち
        a = NULL;
        forall aa in n の ArcList  {
            if( aa のイベント==E && aa の条件が真 ) {
                a = aa;
                break;
            }
        }
    }
    return E;
}

main() {
    - - -
    E = DoSTD(A0); /* 全体の開始アーク A0 から状態遷移の実行を開始 */
}
```

３－５

[3_5_1.c 参照]

[3_5_2.c 参照]

４－１

① 呼ぶ側でチェック

利点：正しい引数値の範囲でしか実行しないような使い方をしている場合や、呼ぶ側での引数のチェックが呼ぶ側の処理で自明になっている場合は、呼ぶ側でのチェックで新たなコードの必要が無く、実行効率もよい。

欠点：いろいろな呼び方をするとき、呼ぶ毎にチェックが必要になる。また、その関数をいろいろなプログラムで共通に使う場合は、チェック漏れが生じやすい。

② 呼ばれる側でチェック

利点：ライブラリ関数など、どこから使われるか分らない場合はチェック漏れを防ぎ、信頼度が高まる。どんな使われ方にも対応できるので、より防衛的である。

欠点：あるプログラムに専用の関数で、その使われ方も決まっている場合、ありえないチェックをして効率が悪くなる場合もある。

③ 両方でチェック

利点：最も防衛的で信頼度が高い。関数を使う側と作る側のプログラマが独立して作業を進める場合は、両方でのチェックが望ましい。テスト完了後、効率アップの改造時にどちらかのチェックを外せばよい。

欠点：実行効率が悪い。但し、どうしても効率が必要であれば、効率アップの改造時にチェックを外せばよい。

４－２　Ｃ言語の例。

```
((year%4==0)&&(year%100!=0))||(year%400==0)
```

４－３

① 恒真

② ｎが 10、20 なら偽、それ以外は真

③ ｎが 10、20 なら真、それ以外は偽

④ 恒偽

４－４

① fp3 ファイルのクローズ

② fp3 ファイルオープンのエラー表示

③ fp2 ファイルのクローズ

④ fp2 ファイルオープンのエラー表示

⑤ fp1 ファイルのクローズ

⑥ fp1 ファイルオープンのエラー表示

4－5

① すべての変数名はその意味を反映した長い名前にすべきである。
反論：内部変数や繰返し制御変数などの意味は、プログラムの文脈で理解できるので名前は i などの簡単なものがすっきりして分りやすい。また、関数名などは、意味を反映した名前にすべきであるが、プログラムの読みやすさを考えてあまり長すぎない方がよい。むしろ名前付け規則を定め、短い名前に十分な意味を含めるようにすべきである。

② プログラムの理解のため、すべての文にはコメントをつけるべきである。
反論：プログラムの理解は、プログラムのコードそのものを読むことが原則であって、コメントはあくまでもコードを読むための補助である。従って、コードを読めば自明であることまでコメントを記述すると、プログラムはかえって分りにくくなる。必要な部分(例えば関数の先頭、重要な条件の意味など)のみにコメントをつけるべきである。また、コメントを付けすぎると、プログラムの修正時にコードのみを修正してコメントの修正を忘れがちになり、コードと合わないコメントになってしまうことが多く、コメントがプログラムの理解を妨げることになってしまう。

5－1

① トップダウンプログラミング
利点：　概要から詳細に向かって、徐々に詳細化していく方法に適している。
欠点：　共通処理を見つけにくい。詳細レベルを合わせにくいことがある。最もリスクの多い部分の決定を後回しにしてしまう恐れがある。

② ボトムアッププログラミング
利点：　共通処理を見つけやすい。
欠点：　最初に詳細な検討に入ってしまい、全体の概要、目標を見失いがちになる。最後になって、もともとの目的に合わないシステムができてしまう恐れがある。

5－2

① 引数や戻り値によるデータ受け渡し
利点：　より明示的で簡単である。変更の影響は呼ぶ方と呼ばれる方の関数にのみ伝播するので、変更に強い。
欠点：　広範囲で使うデータを引数で扱うと、すべての関数の引数としてそのデータを指定する必要があり、煩雑になる。

② 外部変数によるデータの受け渡し
利点：　広範囲で使うデータの場合、引数の場合ような宣言の煩雑さがない。
欠点：　関係が暗黙的であり、わかり難いことが多い。データの変更の影響が広範囲

に伝播するので、変更に弱い。

５－３

① 静的変数：変数領域がプログラム全体の実行開始前に確保され、初期値も実行開始前に１度だけ設定される。関数から戻って、再び関数に入るとき、以前の値が保存されている。

② 動的変数：変数領域は、関数が呼ばれる毎に確保され、初期値も呼ばれる毎に設定されるので、一旦関数を出て再び関数に入ったときには以前の値は消えている。

③ 静的変数を使った場合の問題点：　関数を呼ぶ毎に、実行結果が異なるようになるので、その関数がこれまでにどこからどのように呼ばれたかという履歴を知っておく必要がある（関数の独立性が低くなる)。

５－４

[演習解答図５－４参照]

５－５　　例えば、

① 登録指令のデータ辞書：　登録指令＝書籍ＩＤ＋コピー番号＋棚番号＋登録年月日

② 登録処理：

１．書籍ＩＤをチェックしコピー番号を設定

１．１　新しい書籍かどうかを調べる

１．２　新しい書籍でなければコピー番号を設定(その書籍ＩＤのコピー番号＋１)

１．３　新しい書籍ならコピー番号＝０とする

２．棚番号の生成

２．１　新しい書籍なら、書籍の分類をもとに、棚番号を生成する。

２．２　そうでなければ棚番号はもとの書籍と同じにする。

３．書籍データへ登録

３．１　新しい書籍なら、それを書籍データに登録

３．２　そうでなければ、もとの書籍のコピー番号を増やしたデータを登録

５－６

[演習解答図５－６参照]

５－７

① コントローラ追加型

② 中央変換主導型

③ 入力主導＆変換型

④ 出力主導＆変換型
⑤ 入力主導型：
⑥ 出力主導型：
として、Ｂの処理について検討する。

（１） Ｂが１個の入力データに対して１個の出力データを出す場合、①、③、④が適する。さらに、Ａが複雑な処理が必要なら③、Ｃが複雑な処理が必要なら④が、そのどちらでもなければ①が適切である。

（２） Ｂが１個の入力データに対してｎ個（不定個）のデータを出力する場合は⑤が適切。

（３） Ｂがｎ個（不定個）のデータを入力して１個のデータを出力する場合は⑥が適切。

（４） Ｂがｎ個（不定個）のデータを入力してm個（不定個）のデータを出力する場合は②が適切。

例：「未完」

５－８

例えば、

偶然的強度：本日の新宿駅利用客（たまたま偶然にその日新宿駅を利用しただけ）

論理的強度：ある病院の患者たち
（それぞれ異なった病気を治すための治療を、それぞれ毎に受けている）

時間的強度：ある結婚式場の新郎新婦たち（結婚生活の初期処理の人たちの集まり）

手順的強度：野球チームの３番４番５番の打者（クリーンアップ）
（全体の攻撃の流れの中の一部）

順次的強度：流れ作業の製造チーム
（ある人が作成した製品部分が次の人に部品として使われる）

情報的強度：家族（家族の歴史、文化を共有）

機能的強度：野球チーム（勝つことに向かって一丸となって活動）

５－９

① 内容結合：　言語などでは表現できないが、例えば、あるモジュールの処理の途中に別のモジュールから飛び込む場合など（変数はすべて共有とする）。

② 共有結合：　ＦＯＲＴＲＡＮのＣＯＭＭＯＮブロックなどで、モジュールで利用する変数の領域全体を他のモジュールからも利用できるようにする。Ｃ言語では、例えば、関数の内部変数全体を１つの静的な構造体変数にまとめ、外部にその構造体へのポインタを知らせて、他の関数から構造体の内部を見ることができるようにするなど。

③ 外部結合：　外部変数（グローバル変数）による関数間のデータのやり取り。Ｃ言語での数学関数の”errno”等にも使われる（エラーの結果が入る）

④ 制御結合：　引数の値によって処理が異なる。fopen の第２引数（オープンモード指定）など。この場合、文字列ではあるが、”r”(read mode),”w”(write mode),”a”(append mode)など、決められた値しか渡せない。

⑤ スタンプ結合：　構造体領域全体へのポインタを引数や戻り値で受けわたす。fopen の戻り値や fgetc の引数などで使われるファイルポインタがこの例である。ファイルポインタの宣言での”FILE”は、typedef struct xxxxx　FILE; という宣言が<stdio.h>になされており構造体を指している。但し、この例では、ファイルを利用する側にとっては、この構造体の内容を知る必要はない（これは、「７．２　抽象データ型」の概念にもなっている）。

⑥ データ結合：　標準関数のほとんどの引数や戻り値はデータ結合の例である。例えば、fgetc の戻り値、fputc の第１引数（出力文字）、sin(x),cos(x),tan(x)などの x。

５－１０

[プログラムリスト参照（5-10.c）]

５－１１

```
// recursive tree search
TREE *treeSearch(TREE *root, ELEM key) {
    TREE *pt, *res;
    if( root->elem == key ) return root;
    for(pt=root->fstSon; pt; pt=pt->nextBro)
        if( res=treeSearch(pt) ) return res;
    return NULL;
}

// recursive tree deletion
void treeDel(TREE *root) {
    TREE *pt;
    if(root==NULL) return;
    for(pt=root->fstSon; pt; pt=pt->nextBro)
        treeDel(pt);
    free(root);
}
```

５－１２

[5_12.c　参照]

６－１

起動に対応したデータを用意し、起動の代りにそのデータを作り出せばよい。例えば、図６－１では、「毎日 21 時」データと「毎月 1 日 18 時」データを準備し、毎日 21 時になったときに「毎日 21 時」データを生成する。すると、チェックジョブが起動する。チェックの起動が終われば、「毎日 21 時」データを消去する。「毎月 1 日 18 時」データも同様。

６－２

① ジョブフローによるプログラムの場合

利点：　個々の要素プログラムが独立性が高く簡単であるか、既存のもの（ソート等）が使える。全体としても、簡潔で理解しやすい。

欠点：　より細やかな制御（例えばエラーの場合にその場で修正する等）がやりにくい。また、ジョブ間のデータの受け渡しをファイルで行うため、実行効率が悪くなる。

② 全体を１つのプログラムにまとめた場合

①の反対

６－３

データフローの場合：　ファイルが順アクセスで、かつ２つのジョブ間のデータの受け渡しとしてしか使わないとき（そのデータを読込むジョブが終了すれば、そのデータを消去してもよいとき）。

蓄積データの場合：　ファイルが乱アクセスのとき。または、そのファイルを２つのジョブ間のデータの受け渡し以外にも使うとき（そのデータを読込むジョブが終了したあとで、別のジョブが再度読込む可能性があるとき）

６－４

バッファ容量０：Ada のランデブ（rendezvous）のように、窓口での手渡しによるデータのやり取りに対応。

バッファ容量無限大：十分なファイルスペースのあるシステムでの順次ファイル（sequential file）に相当する。

６－５

中間データのバッファ容量が変わっても、２プロセス間の実行の移り変わり方が変わる（一般に容量が大きいほど、プロセスの実行の移り変わりが少なくなる）だけで、それぞ

れのプロセスの実行履歴は変化しない。処理結果は、それぞれのプロセスの実行履歴によって決まるので、中間データのバッファ容量が変わっても結果は変わらない。

６－６

左方式は、同じバッファを独立した２つのプロセスから読むため、生成側プロセスが生成したデータのある部分は消費側プロセス１が、別の部分は消費側プロセス２が読込んで処理する。このため、２つの消費側プロセスのスケジュールのされ方や実行速度によって、プロセスが読込むデータが異なってくるので、一般にはスケジュールのされ方によって異なった結果が生じる。

一方右の方式では、生成側プロセスは同じデータを２つの消費側プロセスに送るため、２つの消費側プロセスはいつも同じデータ列を入力していくことになる。この場合、スケジュールのされ方によらず、最終的な結果はいつも同じとなる。

６－７

パイプ方式

[6_7_1.c 参照]

[6_7_2.c 参照]

[6_7_3.c 参照]

[6_7_4.c 参照]

[6_7_5.c 参照]

統合例

[6_7.c 参照]

６－８

[6_8.txt 参照]

７－１

例えばテレビセットで：

本質的な情報：　　電源スイッチ、チャネル指定、音量設定

本質的でない情報：ゆがみ補正、輝度調節、コントラスト調節など

（輝度やコントラストは、テレビセットが適切に設定してくれれば、見る方はそれほど気にしなくてもよいので、本質的でない情報にした）

さらに、ビデオ付きのテレビの場合は、上に加えて、

本質的な情報：　　テープセット／取出し、再生、録画、早送り、巻戻し、停止など

本質的でない情報：テープ回転数指示、再生レベル／録画レベルの指定など、

７－２

ビデオの場合：

上級者向け本質的情報：一時停止、スロー再生、録画チャネル指定(見ているチャネル以外の録画のときに指定)、次の録画までスキップなど。

修理する人向け：テープ回転速度、再生レベル／録音レベル、モータ消費電力、モータ温度など（ビデオの修理をしたことがないため、あくまで想像です）

７－３

例えば、外部変数 x に対して、

```
int x=0;
```

の宣言の代りに、

```
int x(int mode, int val) {
    static int x=0;
    if(mode==1) x=val;
    return x;
}
```

のような関数 x を宣言し、

```
x=xxxx;/*値の設定*/   y=x;/*値の参照*/
```

の代りに、以下のような関数呼び出しを使う。

```
x(1,xxxx);/*値の設定*/ y=x(0,0);/*値の参照*/
```

これによって、外部変数はすべて関数に隠すことができる。

または、

```
int x=0;
int getx() { return x; }
void setx(int val) { x=val; }
```

として、上記の x の値の設定と参照を、以下のように代える。

```
 setx(xxxx); );/*値の設定*/ y=getx();/*値の参照*/
```

この場合は、x は getx と setx でしか使われない。

７－４

成績データの登録の仕方、検索の仕方によって、適切な操作集合は異なる。ここではある前提（コメント部分参照）をもとにした操作の例を掲げる。

1.テーブル新規生成

/* 学生の登録は入学年度と学科毎に学科事務担当が学生情報を登録する。*/

2.学生登録開始(入学年度、学科)

3.学生登録（学生番号、氏名）

4.学生登録終了

/* 履修科目はセメスタ毎に学生が申告登録する。*/

- 履修科目登録開始(学生番号、セメスター)
- 履修科目登録(科目)
- 利主科目削除(科目)
- 履修科目登録終了

/* 成績は、セメスター科目毎に、担当教員が登録する。*/

- 成績登録開始(セメスター、科目)
- 成績登録(学生番号、点数)
- 成績変更(学生番号、点数)
- 成績登録終了

/* 学生個人毎の学生情報を検索する(事務課他が利用) */

- 学生情報検索(学生番号)→氏名、入学年度、学科
- 学生履修科目一覧開始(学生番号、セメスター)
- 学生履修科目読取→科目／終り

/* 学生、セメスターごとに科目と成績の一覧を取出す(学生、学科担任が利用) */

- 学生成績読出開始(学生番号、セメスター)
- 学生成績読出→科目、成績／終り
- 学生科目成績検索(学生番号、セメスター、科目)→成績

/*科目とセメスター[と学科]ごとに学生と成績の一覧を取出す(科目担当教員が利用)*/

- 科目成績一覧開始(科目、セメスター、[学科])
- 科目成績読取→学生番号、成績／終り

（注：成績登録まえには、これは科目、セメスター毎の履修者一覧読取として利用できる。）

７－５

三角形ＡＢＣの重心は、ＡとＢＣの中点を結ぶ線と、ＢとＡＣの中点を結ぶ線の交点であるから、例えば次のようなプログラムになる。

```
点  A, B, C, P1, P2, O;
線  L1, L2;
三点の取出し(&A,&B,&C); /* A,B,C の値を求める */
```

```
L1=線の生成(A, 中点(B,C));
L2=線の生成(B, 中点(A,C));
O=２線の交点(L1,L2);
重心 O の表示;
```

７－６

クラスとインスタンスの関係は、プログラムの実行により動的に定められるのに対して、汎化クラスと継承クラスの関係は、実行前に（プログラム作成時に）静的に定められる。。

７－７

① 学生と履修科目：関連。関連の情報として点数などが入る。
② 履修科目と「プログラミング入門」：クラスとインスタンス
③ 履修科目と、必須科目、選択科目：汎化と継承(履修科目の継承クラスが必須／選択科目)
④ 履修科目と担当教員：関連
⑤ 学部と学科：集約（学科 is-a-part-of 学部といえる）
⑥ 学科と学生：一般には集約。関連のこともある。

７－８

① 学生の卒業に必要な残りの単位数：学生クラス
② 履修し合格したときの取得単位数：科目クラス
③ 学生の科目毎の成績(点数)：履修関係クラス
④ 科目平均点：科目クラス
⑤ 履修者内の成績順位：履修関係クラス

７－９

・データ抽象化による方式：
利点：テーブルに利用の仕方に応じた適切な操作を設定できる。また、実現の仕方については構造体とポインタによるリストを使ったり、ライブラリを利用したり、必要に応じて自由に選択できる。
欠点：操作の選び方に個人差が出て、常に理解しやすい抽象化がえられるとは限らない。実現方法も自由度がありすぎて、かえって複雑になる恐れもある。一般に、このような問題毎のデータ抽象化方式を用いると、それを利用する側のプログラムは簡単になるが、抽象データの実現が複雑になり、難しさが抽象データの実現にしわ寄せされがちとなるので注意が必要である。
・クラス関係による方式

利点：データの利用の仕方が少々変わっても、クラス構造自体は変えないでそれを使う側でいろいろ対処しやすい。（テーブルデータ抽象化と比べると）問題によらないで、比較的個人差のない標準的な設計ができる。複雑な処理も、ライブラリにまかせるようなプログラムになることが多い。
欠点：実行効率が問題になる場合がある（その場合は、適切に処理や関連を省略していく必要がある）。問題特有の使い方に沿った効率化がしずらい。
（一般には、クラス関係による方式を利用する方がよいことが多い）

７－１０

図 7.13 を構造体とポインタの関係で表わす。図の vector(rv)はリスト構造で表わすとする。つまり、ノードは rellist の先頭を持ち、rellist は rel へのポインタをもつ。構造体の宣言は以下のようになる。

```
struct Anode {
    - - -
    struct rellist *relHead;
};
struct Bnode {
    - - -
    struct rellist *relHead;
};
struct rellist {
    struct rellist *next;
    struct rel *relp;
}
struct rel {
    struct Anode *anode;
    struct Bnode *bnode;
    RELDATA  reldata;
}
```

もし、ＡもＢも同一タイプのノードであり、関係の方向性が必要であれば、

```
struct node {
    - - -
    struct rellist *srcHead, *dstHead;
}
```

```
struct rellist {
    struct rellist *next;
    struct rel *relp;
}
struct rel {
    struct node *srcnode, *dstnode;
    RELDATA  reldata;
}
```
となる。

ここで、rellist と rel をまとめてひとつの構造体にすることができる。つまり、rel は常に２つのリスト(src のリストと dst のリスト)の要素になっているので、これら 2 つのリストのポインタ（*srcnext, *dstnext）を rel が持つようにする。この結果、つぎのような上と等価な構造体が得られ、rellist をなくすことができる。

```
struct node {
    - - -
    struct rel *srcHead, *dstHead; ;  /* 必要なら Tail ももつ */
}
struct rel {
    struct rel *srcnext, dstnext;
    struct node *srcnode, *dstnode;
    RELDATA  reldata;
}
```

上の宣言の node に*srcTail, *dstTail を加えると、図 3.12 の有向グラフの宣言と等価になる。

７－１１

抽象クラス：オブジェクトを生成できない。操作の実現プログラムを持たない。

具象クラス：オブジェクトを生成できる。操作の実現プログラムを持つ。

抽象クラスの必要性：

抽象クラスは、インタフェースの定義と、そのインタフェース操作を下位のいくつかの具象クラスに継承するために使われる。共通的な抽象クラスのインタフェース操作を呼出すことにより、具象クラスの種類を区別することなく、適切に継承された操作を実行できる。「８．１　デザインパターン」の例を参照のこと。

７－１２

抽象クラスを使ったプログラムのイメージ：

抽象クラス 学生

抽象操作　授業料請求

具象クラス 特待生：学生

授業料請求：何もしない

具象クラス　派遣学生：学生

属性　＜授業料、派遣元企業＞

授業料請求：請求(あて先：派遣元企業、名目：委託金、金額：授業料)；

具象クラス　一般学生：学生

属性　＜授業料、親元＞

授業料請求：請求(あて先：親元、名目：大学授業料、金額：授業料)；

具象クラス　留学生：学生

属性　＜授業料＞

授業料請求：請求(あて先：本人、名目：大学授業料、金額：授業料)；

```
listof(学生) ST=---;
```

全学生の授業料請求のプログラム：

```
forall s in ST {
    s.授業料請求
}
```

抽象クラスを使わない場合

具象クラスのプログラムは上と同様。全学生の授業料請求のプログラムが以下のようになる。

```
vector ST=---;    /* JAVA の vector はいろいろなクラスの集合として使える */
```

全学生の授業料請求のプログラム：

```
forall s in ST {
    if(s のクラスが特待生)
            s.特待生の授業料請求
    else if(s のクラスが派遣学生)
            s. 派遣学生の授業料請求
    else if(s のクラスが一般学生)
            s. 一般学生の授業料請求
    else if(s のクラスが留学生)
            s. 留学生の授業料請求
}
```

８－１

[演習解答図８－１参照]

８－２

未完

８－３

デザインパターンはオブジェクト指向によるプログラム設計において、クラスの構成の仕方に関するノウハウをパターン化して集めたものであり、そのパターンに当てはまるプログラムの部分に適用できる。

一方、アーキテクチャパターンは、プログラムの部分ではなく、システム全体が、どのような構成要素のどのような関連によってできているかを示したもので、より大きな単位を扱う。

デザインパターンもアーキテクチャパターンも、プログラムやシステムの設計におけるノウハウをパターンとしてまとめたものであり、共に、要素と要素間関連の形についまとめている。

８－４

例えば UNIX カーネルの TCP/IP スタックのプログラムを調べてみる。（上級問題）

９－１

[構造に基づくテスト]

利点：比較的、客観的なテストができる。テストケースの生成を自動化（またはシステマティックに）できる。

欠点：必要な機能の抜け、機能の誤りなどのテストができない恐れがある。

[機能に基づくテスト]

利点：プログラムが仕様通りかどうかのテストができる。

欠点：仕様の捉え方が難しい。テストケースの選び方が主観的になりやすい。

９－２

「９．２　構造に基づくテスト」参照のこと。

９－３

制御フロー図は「演習解答図９－３参照」

全パス（X:不可能なパス）

1. 1-E

   例えば、from=0, to=0, data[]={1}。

2.(X)1-2-3-4-14-15-E

   from<to で left=from;right=to;だから left<right。よって 4-14 はない。

3.(X)1-2-3-4-5-7-9-4-14-15-E

   4-5-7-9 なら、left<=right のまま。よって 9-4 はありえない。

4.(X)1-2-3-4-5-6-7-9-4-14-15-E

   from<to より 4 では left<right。その後 4-14 になるには、left>right になる必要があり、left++か right の実行が最低２回必要。よってありえない。

5.(X)1-2-3-4-5-7-8-9-4-14-15-E

   4 と同様の理由でありえない。

6.(X)1-2-3-4-5-6-7-8-9-4-14-15-E

   この場合、6 と 8 の実行で、left<right から left>right になる。このためには、to-from=1(データ個数２)でなければならない。そうすると、pivot=data[left]となり、5-6 はありえない。

7. 1-2-3-4-5-7-9-10-11-12-13-4-14-15-E

   例えば、from=0, to=1, dtat[]={5,3}

8. 1-2-3-4-5-6-7-9-10-11-12-13-4-14-15-E

   例えば、from=0, to=2, data[]={1,5,3}

9. 1-2-3-4-5-7-8-9-10-11-12-13-4-14-15-E

   例えば、from=0, to=2, data[]={3,5}

10. 1-2-3-4-5-6-7-8-9-10-11-12-13-4-14-15-E

    例えば、from=0, to=2, data[]={1,3,5}

９－４

「演習解答図９－４参照」

９－５

「演習解答図９－５参照」

９－６

MaCabe の測度は、演習解答図９－３参照。（結果は６）

Halstead の測度の計算：