言語処理プログラミング 課題１

（１）課題名：字句の出現頻度表の作成
（２）演習期間：9/28 - 10/18
（３）レポート・プログラム提出期間：10/19 - 10/26

（４）課題

プログラミング言語 MPPL で書かれたプログラムらしきものを読み込み，字句（トークン）がそれぞれ何個出現したかを数え，出力する C プログラムを作成する．例えば，作成するプログラム名を tc，MPPL で書かれたプログラムらしきもののファイル名を foo.mpl とするとき，コマンドラインからのコマンドを

tc foo.mpl

とすれば(foo.mpl のみが引数)，foo.mpl 内の字句の出現個数を出力するプログラムを作成する．

入力：MPPL で書かれたプログラムらしきもののファイル名．コマンドラインから与える．字句（トークン）は，名前，キーワード，符号なし整数，文字列，記号のいずれかである．ただし，キーワードと記号については，それぞれの記号列が別々の字句であるが，名前，符号なし整数，文字列については，その実体が異なっていても同じ「名前」,「符号なし整数」,「文字列」という字句であるとする．字句の定義として，MPPL のプログラムのマイクロ構文を次のように与える（左辺の括弧の中は非終端記号の英語表記であり参考のために付加してある）．なお，終端記号は ASCII 文字である．

プログラム(program) ::= { 字句 | 分離子 }
字句(token) ::= 名前 | キーワード | 符号なし整数 | 文字列 | 記号
名前(name) ::= 英字 { 英字 | 数字 }
キーワード(keyword) ::= "p" "r" "o" "g" "r" "a" "m" | "v" "a" "r" | "a" "r" "r" "a" "y" | "o" "f" |
"b" "e" "g" "i" "n" | "e" "n" "d" | "i" "f" | "t" "h" "e" "n" | "e" "l" "s" "e" | "p" "r" "o" "c" "e" "d" "u" "r" "e" |
"r" "e" "t" "u" "r" "n" | "c" "a" "l" "l" | "w" "h" "i" "l" "e" | "d" "o" | "n" "o" "t" | "o" "r" | "d" "i" "v" |
"a" "n" "d" | "c" "h" "a" "r" | "i" "n" "t" "e" "g" "e" "r" | "b" "o" "o" "l" "e" "a" "n" |
"r" "e" "a" "d" | "w" "r" "i" "t" "e" | "r" "e" "a" "d" "l" "n" | "w" "r" "i" "t" "e" "l" "n" |
"t" "r" "u" "e" | "f" "a" "l" "s" "e" | "b" "r" "e" "a" "k"
符号なし整数(unsigned integer) ::= 数字 { 数字 }
文字列(string) ::= "'" { 文字列要素 | "'" "'" } "'"
// "'"はアポストロフィ（シングルクォート）である
記号(symbol) ::= "+" | "-" | "*" | "=" | "<" ">" | "<" | "<" "=" | ">" | ">" "=" |
"(" | ")" | "[" | "]" | ":" "=" | "." | "," | ":" | ";"
英字(alphabet) ::= "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" |
"n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" |
"A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" |
"N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z"
数字(digit) ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

分離子(separator) ::= 空白 | タブ | 改行 | 注釈
注釈(comment) ::= "{" { 注釈要素 } "}" | "/" "*" { 注釈要素 } "*" "/"

ただし,
・文字列要素(string element)：アポストロフィ"'",改行以外の任意の表示文字を表す,
・文字列には長さの概念があり，その文字列に含まれるアポストロフィ"'",改行以外の任意の表示文字の数と 連続する"'" "'"の組の数の和である．即ち"'"については連続する 2 つを 1 と数える（同様に連続する 4 つを 2 と数える．以下同じ)．言い換えると，文字列のマイクロ構文での{ }の繰り返し回数が文字列の長さである,
・注釈要素(comment element)：注釈が"{","}"で囲まれているときは，注釈要素は閉じ中括弧"}"以外の任意の表示文字を表し，"/"'"*","*"'"/"で囲まれているときは，連続する注釈要素として"*"'"/"が現れないことを除いて任意の表示文字を表す,
・空白(space)，タブ(tab)は，ASCII コードではそれぞれ 20, 09(16 進数表示)であり，C 言語上では，' ', '¥t' で表現される,
・改行(end of line)は OS によって異なるので，'¥r', '¥n', '¥r' '¥n', '¥n' '¥r'の 4 通りの ASCII コード(列)を一つの改行として扱うこと．ただし，'¥r', '¥n'は ASCII コードではそれぞれ 0D, 0A(16 進数表示)である,
・表示文字(graphic character)とは，タブ，改行と通常画面に表示可能な文字(ASCII では 20 から 7E(16 進数表示)までの文字)を意味する,
・表示文字ではない文字コード(タブ，改行以外の制御コード)が現れた場合は，存在しないものとして無視してよい．もちろん，エラーとしてもよい,
・制約規則としては,
　　最長一致規則：字句として，二つ以上の可能性があるときは最も長い文字の列を字句とする,
　　英大文字と小文字は区別する,
　　キーワードは予約されている：キーワードは名前ではない,
がある.
・なお，このファイルは ASCII コードによるテキストファイルであるとしてよい．あまりにも長い行（例えば，1 行 1000 文字以上など）はないものとしてよいが，あったとしても，作成したプログラムが実行時エラーを起こしてはいけない.

［注意］構文において，用いられている記号の意味は次の通りである.

" "	終端記号であることを示す
::=	左辺の被終端記号が右辺で定義されることを示す
|	左側と右側のどちらかであることを示す
{ }	内部を 0 回以上繰り返すことを表す
[ ]	内部を省略してもよい(0 回か 1 回)ことを示す

出力：字句とその出現数の表．標準出力へ出力する．たとえば,

"and"　　10
"array"　　2
・・・・・
"NAME"　　38

のように出力せよ．ただし，名前，文字列，符号なし整数はその実体が異なっていてもそれぞれを同じ字

句として扱い，出現しない字句については出力しない（出力中に個数 0 の行はない）．なお，分離子は字句ではないので，注釈などについては出力する必要はない．また，字句や分離子を構成しない文字が現れたとき（コンパイラとしてはエラーである）は，その旨（エラーメッセージ）を標準出力へ出力し，ファイルの先頭からそこまでの部分について，出現数の表を出力せよ．下記の字句解析系がエラーを検出した場合も同様である．

（５）プログラム作成条件

入力から，字句を切り出す字句解析系(スキャナ)と，字句を数え，表を作成する主手続きとに分割して作成せよ．字句解析系は後の課題で再利用するため，以下のようなモジュール仕様とせよ．

(5-1)初期化関数　int init_scan(char *filename)

filename が表すファイルを入力ファイルとしてオープンする．

戻り値：正常な場合：0 を返し，ファイルがオープンできない場合など異常な場合は負の値を返す．

(5-2)トークンを一つスキャンする関数　int scan()

次のトークンのコードを返す．トークンコードは別ファイル(token-list.h)参照のこと．

End-of-File 等次のトークンをスキャンできないとき，戻り値として負の値を返す．

(5-3)定数属性　int num_attr;

scan()の戻り値が「符号なし整数」のとき，その値を格納している．

なお，32767 よりも大きい値の場合は，エラーである．

(5-4)文字列属性　char string_attr[MAXSTRSIZE];

scan()の戻り値が「名前」または「文字列」のとき，その実際の文字列を格納している．また，それが「符号なし整数」のときは，入力された数字列を格納している．その文字列(数字列，名前)は，'¥0'で終端されている．例えば，文字列が'It"s'のときには，string_attr には先頭から順に，'I', 't', '¥', '¥', 's', '¥0'が格納される．

もし，string_attr に格納できないくらい長い文字列（数字列，名前）の場合には，エラーである．

(5-5)行番号関数　int get_linenum()

もっとも最近に scan()で返されたトークンが存在した行の番号を返す．まだ一度も scan()が呼ばれていないときには 0 を返す．

(5-6)終了処理関数　void end_scan()

init_scan(filename)でオープンしたファイルをクローズする．

ただし，必要に応じてこれら以外のインタフェース関数を用意してもかまわない．

（６）例

(6-1)入力ファイル例

```
program sample11pp;
procedure kazuyomikomi(n : integer);
begin
        writeln('input the number of data');
        readln(n)
end;
```

```
        var sum : integer;
        procedure wakakidasi;
        begin
                writeln('Sum of data = ', sum)
        end;
        var data : integer;
        procedure goukei(n, s : integer);
                var data : integer;
        begin
                s := 0;
                while n > 0 do begin
                        readln(data);
                        s := s + data;
                        n := n - 1
                end
        end;
        var n : integer;
begin
        call kazuyomikomi(n);
        call goukei(n * 2, sum);
        call wakakidasi
end.
```

(6-2)プログラム sample11pp.mpl に対する出力例

```
        "NAME     "       27
        "program  "        1
        "var      "        4
        "begin    "        5
        "end      "        5
        "procedure "       3
        "call     "        3
        "while    "        1
        "do       "        1
        "integer  "        6
        "readln   "        2
        "writeln  "        2
        "NUMBER   "        4
        "STRING   "        2
        "+        "        1
        "-        "        1
        "*        "        1
        ">        "        1
        "(        "        8
        ")        "        8
        ":=       "        3
        ".        "        1
        ",        "        3
        ":        "        6
        ";        "       17
```

（7）拡張仕様

もし，余裕があれば，名前についてはその実体ごとにも出現個数を数えて出力するように拡張せよ．つ

まり，n, sum 等の名前毎にも出現個数を数えて出力せよ.

（8）スキャナ用ヘッダファイル

```
/* token-list.h  */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define MAXSTRSIZE 1024

/* Token */
#define       TNAME         1      /* Name : Alphabet { Alphabet | Digit } */
#define       TPROGRAM      2      /* program : Keyword */
#define       TVAR          3      /* var : Keyword */
#define       TARRAY        4      /* array : Keyword */
#define       TOF           5      /* of : Keyword */
#define       TBEGIN        6      /* begin : Keyword */
#define       TEND          7      /* end : Keyword */
#define       TIF           8      /* if : Keyword */
#define       TTHEN         9      /* then : Keyword */
#define       TELSE         10     /* else : Keyword */
#define       TPROCEDURE    11     /* procedure : Keyword */
#define       TRETURN       12     /* return : Keyword */
#define       TCALL         13     /* call : Keyword */
#define       TWHILE        14     /* while : Keyword */
#define       TDO           15     /* do : Keyword */
#define       TNOT          16     /* not : Keyword */
#define       TOR           17     /* or : Keyword */
#define       TDIV          18     /* div : Keyword */
#define       TAND          19     /* and : Keyword */
#define       TCHAR         20     /* char : Keyword */
#define       TINTEGER      21     /* integer : Keyword */
#define       TBOOLEAN      22     /* boolean : Keyword */
#define       TREADLN       23     /* readln : Keyword */
#define       TWRITELN      24     /* writeln : Keyword */
#define       TTRUE         25     /* true : Keyword */
#define       TFALSE        26     /* false : Keyword */
#define       TNUMBER       27     /* unsigned integer */
#define       TSTRING       28     /* String */
#define       TPLUS         29     /* + : symbol */
#define       TMINUS        30     /* - : symbol */
#define       TSTAR         31     /* * : symbol */
#define       TEQUAL        32     /* = : symbol */
#define       TNOTEQ        33     /* <> : symbol */
#define       TLE           34     /* < : symbol */
#define       TLEEQ         35     /* <= : symbol */
#define       TGR           36     /* > : symbol */
#define       TGREQ         37     /* >= : symbol */
#define       TLPAREN       38     /* ( : symbol */
#define       TRPAREN       39     /* ) : symbol */
#define       TLSQPAREN     40     /* [ : symbol */
```

```
#define       TRSQPAREN     41      /* ] : symbol */
#define       TASSIGN       42      /* := : symbol */
#define       TDOT          43      /* . : symbol */
#define       TCOMMA        44      /* , : symbol */
#define       TCOLON        45      /* : : symbol */
#define       TSEMI         46      /* ; : symbol */
#define       TREAD         47      /* read : Keyword */
#define       TWRITE        48      /* write : Keyword */
#define       TBREAK        49      /* break : Keyword */

#define NUMOFTOKEN   49

/* token-list.c */

#define KEYWORDSIZE  28

extern struct KEY {
       char * keyword;
       int keytoken;
} key[KEYWORDSIZE];

extern void error(char *mes);

/* scan.c */
extern int init_scan(char *filename);
extern int scan(void);
extern int num_attr;
extern char string_attr[MAXSTRSIZE];
extern int get_linenum(void);
extern void end_scan(void);
```

（9）課題1メインプログラム例（サンプル）

```
#include "token-list.h"

/* keyword list */
struct KEY key[KEYWORDSIZE] = {
       {"and",       TAND   },
       {"array",     TARRAY },
       {"begin",     TBEGIN },
       {"boolean",   TBOOLEAN},
       {"break",     TBREAK  },
       {"call",      TCALL  },
       {"char",      TCHAR  },
       {"div",       TDIV   },
       {"do",        TDO    },
       {"else",      TELSE  },
       {"end",       TEND   },
       {"false",     TFALSE },
       {"if",        TIF    },
       {"integer",   TINTEGER},
       {"not",       TNOT   },
```

```
        {"of",        TOF    },
        {"or",        TOR    },
        {"procedure", TPROCEDURE},
        {"program",   TPROGRAM},
        {"read",      TREAD  },
        {"readln",    TREADLN},
        {"return",    TRETURN},
        {"then",      TTHEN  },
        {"true",      TTRUE  },
        {"var",       TVAR   },
        {"while",     TWHILE },
        {"write",     TWRITE  },
        {"writeln",   TWRITELN}
};

/* Token counter */
int numtoken[NUMOFTOKEN+1];

/* string of each token */
char *tokenstr[NUMOFTOKEN+1] = {
        "",
        "NAME", "program", "var", "array", "of", "begin", "end", "if", "then",
         "else", "procedure", "return", "call", "while", "do", "not", "or",
        "div", "and", "char", "integer", "boolean", "readln", "writeln", "true",
         "false", "NUMBER", "STRING", "+", "-", "*", "=", "<>", "<", "<=", ">",
        ">=", "(", ")", "[", "]", ":=", ".", ",", ":", ";", "read","write", "break"
};

int main(int nc, char *np[]) {
        int token, i;

    if(nc < 2) {
        printf("File name id not given.¥n");
        return 0;
    }
    if(init_scan(np[1]) < 0) {
        printf("File %s can not open.¥n", np[1]);
        return 0;
    }
    /* 作成する部分：トークンカウント用の配列？を初期化する */
    while((token = scan()) >= 0) {
        /* 作成する部分：トークンをカウントする */
    }
    end_scan();
    /* 作成する部分:カウントした結果を出力する */
    return 0;
}

void error(char *mes) {
        printf("¥n ERROR: %s¥n", mes);
        end_scan();
}
```

（１０）おまけ(id-list.c)

```
#include "token-list.h"

struct ID {
        char *name;
        int count;
        struct ID *nextp;
}  *idroot;

void init_idtab() {          /* Initialise the table */
        idroot = NULL;
}

struct ID *search_idtab(char *np) { /* search the name pointed by np */
        struct ID *p;

        for(p = idroot; p != NULL; p = p->nextp) {
                if(strcmp(np, p->name) == 0) return(p);
        }
        return(NULL);
}

void id_countup(char *np) {  /* Register and count up the name pointed by np */
        struct ID *p;
        char *cp;

        if((p = search_idtab(np)) != NULL) p->count++;
        else {
                if((p = (struct ID *)malloc(sizeof(struct ID))) == NULL) {
                        printf("can not malloc in id_countup¥n");
                        return;
                }
                if((cp = (char *)malloc(strlen(np)+1)) == NULL) {
                        printf("can not malloc-2 in id_countup¥n");
                        return;
                }
                strcpy(cp, np);
                p->name = cp;
                p->count = 1;
                p->nextp = idroot;
                idroot = p;
        }
}

void print_idtab() {  /* Output the registered data */
        struct ID *p;

        for(p = idroot; p != NULL; p = p->nextp) {
                if(p->count != 0)
```

```
                        printf("¥t¥"Identifier¥" ¥"%s¥"¥t%d¥n", p->name, p->count);
        }
}

void release_idtab() {        /* Release tha data structure */
        struct ID *p, *q;

        for(p = idroot; p != NULL; p = q) {
                free(p->name);
                q = p->nextp;
                free(p);
        }
        init_idtab();
}
```

（１１）言語処理プログラミングを行うための事前計画（スケジュール）と実際の進捗状況について

(11-1)事前計画（スケジュール）

終えるために時間のかかる作業（仕事，タスク）をするときには，スケジュールを立てることが重要である．特に作業結果の質を落とせなくて，〆切も厳密に定められているときに，スケジュールを立てずに作業を行うことは危険過ぎる．

ここでいうスケジュールとは，いつ何を行うかを時刻順に並べて書いたものである．スケジュールがあると次のようなメリットがある．

・日々何を行うかが明確なので迷うことがない．
・何を行うか（作業内容）がわかっているので，その日の作業に必要なものや知識を前もって用意できる．
・現実がスケジュールよりも遅れ始めたときにすぐ気づけて対処できる．〆切前日に遅れに気づいても時間がないので対処できない（〆切を守れない）．

従って，スケジュールを立てるためには，次の情報が必要である．

(a)作業全体をどのような部分作業（サブタスク）や具体的な行動に分割できるか．
(b)各サブタスクの実行の前後関係．例えば，コンパイルが正常に終わらなければテストはできないなど．
(c)各サブタスクを行うのに必要と思われる時間（タスクの見積もり時間）．
(d)この作業に使えない時間帯（睡眠，他の作業を行う時間など）．しかし，これらは，作業の重要度や〆切の切迫度で幾分変動する．

(a)については，作業はできるだけ細かな部分作業に分割するのが望ましい．なぜなら，部分作業が細かく具体的であればあるほど(c)の見積もり時間が正確になるからである．また，(b)を考えることで，部分作業の抜け（考え落とし）を防止する効果もある．

とはいえ，初めて行う作業（今回の演習が該当するかもしれない）の場合は，何をすればよいかがよくわかっていないので，(a)で細かく分割できないかもしれない．その結果(c)での見積もりも甘いものにならざるを得ない．しかし，作業を始めてしまえば，だんだん何をすればよいかがわかってくるものである．従って，再スケジュールが必要となる．再スケジュールは，作業の詳細がわかったとき，その結果，見積

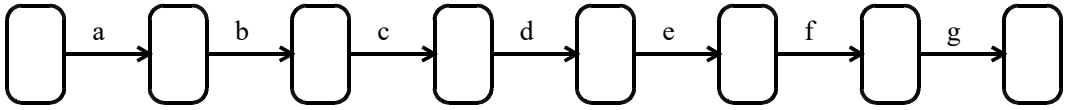
もり時間が間違っていたことがわかったとき,作業の遅れに気づいたときなどに随時行うべきものである．もちろん，再スケジュールを行っても，最後の〆切を守るように再スケジュールしなくてはならない．従って，スケジュールは，最後に予備日を置くなと，余裕を持って立てる必要がある．

(11-2)スケジューリングの例（課題 1）

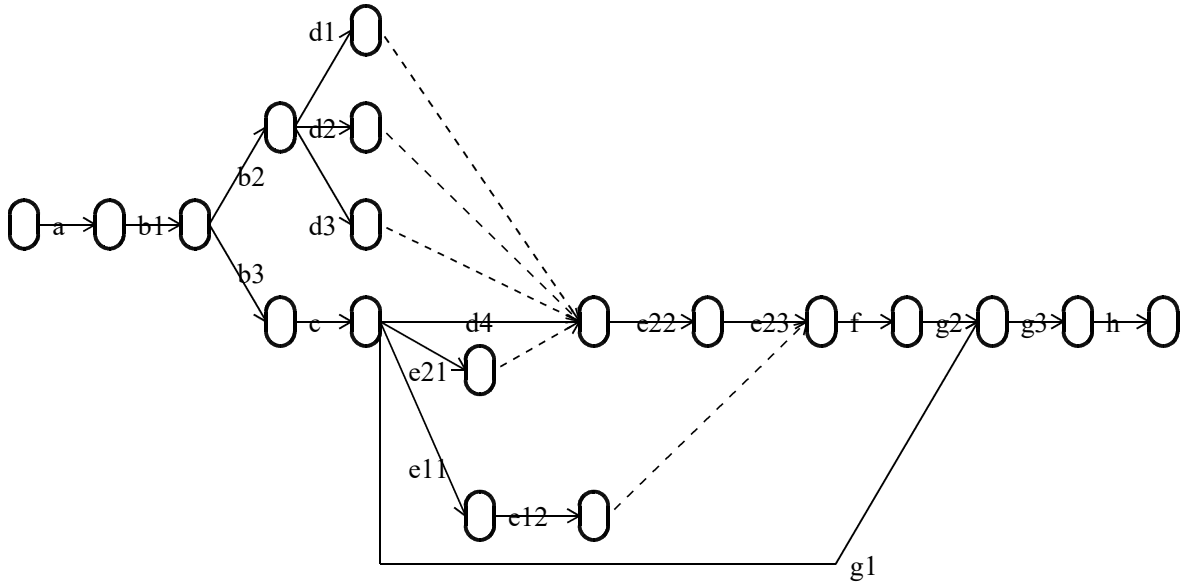
作るべきものの詳細がよくわからないので，まず，本日の説明と配付資料からわかる範囲でスケジュールを立てる．次はその例（見積もり時間なし）である．段付けは部分作業（サブタスク）を表す.

(a)スケジュールを立てる
(b)資料を読む
    (b-1)配布された資料を読み直す
    (b-2)配布されたプログラムを読む
    (b-3)コンパイラのテキスト（特にサンプルコンパイラの字句解析系の部分）を読む
(c)字句解析系（スキャナ）の概略設計（どのような関数が必要かと関数の外部仕様作成）
(d)プログラム作成(コーディング)
    (d-1)(9)のメインプログラム例のトークンカウント用の配列を初期化部分の作成
    (d-2)(9)のメインプログラム例のトークンをカウント部分の作成（スキャナを利用)）
    (d-3)(9)のメインプログラム例のカウントした結果の出力部分の作成
    (d-4)スキャナの作成
(e)テストプログラムの作成
    (e-1)ブラックボックステスト用プログラムの作成
        (e-11)ブラックボックステスト用プログラムの作成
        (e-12)バグがない場合の想定テスト結果の準備
    (e-2)ホワイトボックステスト用プログラムの作成
        (e-21)カバレッジレベルの決定(何パーセントのカバレッジを目指すか)
        (e-22)ホワイトボックステスト用プログラムの作成
        (e-23)バグがない場合の想定テスト結果の準備
(f)テストとデバッグを行う
(g)レポートの作成
    (g-1)作成したプログラムの設計情報
    (g-2)テスト情報
    (g-3)この課題を行うための事前計画（スケジュール）と実際の進捗状況
(h)プログラムとレポートの提出

次にこれらのタスクの関連を図示する．接点がタスクの区切り（サブタスクの開始点や終了点）のポイント，矢印がタスクを表す．従って，グラフの接続関係がタスクの実行の順序制約を表す．このような図の各タスクにタスクを行うのに必要な見積もり時間を付加したものをパート(PERT)図という.



この図は書くまでもない簡単な図であるが，サブタスクレベルまで分解して書くと意味のある図になる．例えば，(e-1)は(c)の後直ちに実行可能である．また，早い段階から開始することができるが完全に終えるには他のタスクが終了しなくては駄目なタスクもあり得る．このようなタスクはさらにサブタスクに分割できることが多い．一般に，次の図は，サブタスクレベルで書いたパート図の例である．



ただし，点線の矢印はダミーのタスクである（２節点間の矢印は高々１本に限るというグラフ理論の制約のため)．パート図はサイクルのない有向グラフになるので，パート図のタスクを順序制約を守って１列に並べることができ，実際の実行順を決定する．

各タスクの見積もり時間を決定（推定）した後，最後に，各タスクに，各タスクの実行開始日（時刻）と実行終了予定日（時刻）を前から順に決定する（これをレポートに書くこと)．このとき，タスク実行のために 1 日 24 時間使えると考えてはいけない．また，最後のタスクの実行終了予定日は〆切日より前でなくてはならないし，もっと言うなら，十分な予備日が必要である．予備日の日数は，作業見積もりの精度が高ければ少なくてもよいし，低ければそれなりの日数を用意するのが必要である．見積もり時間を多く取ると，(〆切が変わらないとすれば）各タスクで使える時間が少なくなる．言い換えると，作業量がよくわからない時は，早めに仕事を進めよ，ということである

万一，実際の進捗がスケジュールよりも遅れた場合，対処法は次の通りである．

・予備日を減らして未実行タスクの実行日を遅らせ，時間を作る．
・未実行のサブタスクの見積もり時間を減らして，時間を作る．
・本来別のことに使う予定であった時間を作業時間に組み込む．例えば，遊びに行くのを止めたり，睡眠時間を減らしたりするなど

いずれにせよ，遅れた理由を究明し，その原因を早急に潰す必要がある．その理由が一般的なものであれ

ば，他の未実行タスクも同様の理由で遅れる可能性もあるので，早めに行動するのが大事である．質問等は，どんな内容でも積極的に行うこと．

事前作業計画の例（日付は過去の例である）

開始予定日	終了予定日	見積もり時間	作業内容
10/3	10/3	1	(a)スケジュールを立てる
10/4	10/4	0.5	(b-1)配布された資料を読み直す
10/4	10/4	0.5	(b-2)配布されたプログラムを読む
10/4	10/4	1	(b-3)コンパイラのテキスト(プログラム)を読む
10/5	10/7	5	(c)字句解析系（スキャナ）の概略設計
10/8	10/8	2	(e-11)ブラックボックステスト用プログラムの作成
10/9	10/11	5	(d-4)スキャナの作成
10/12	10/12	1	(e-12)バグがない場合の想定テスト結果の準備
10/13	10/13	0.5	(d-1)(9)のトークンカウント用の配列を初期化部分の作成
10/13	10/13	0.5	(d-2)(9)のトークンをカウント部分の作成
10/13	10/13	1	(d-3)(9)のカウントした結果の出力部分の作成
10/14	10/14	0.5	(e-21)カバレッジレベルの決定
10/14	10/14	2	(e-22)ホワイトボックステスト用プログラムの作成
10/15	10/15	1	(e-23)バグがない場合の想定テスト結果の準備
10/16	10/20	8	(f)テストとデバッグを行う
10/28	10/28	1	(g-1)作成したプログラムの設計情報を書く
10/29	10/29	1	(g-2)テスト情報を書く
10/30	10/30	1	(g-3)事前計画と実際の進捗状況を書く
10/31	10/31	−	(h)プログラムとレポートの提出

注意:

・レポートの作成(g)は，対応する作業をするときに記録をちゃんと残しておけば，それをコピーするだけですむはず．

・プログラムとレポートの提出予定日が〆切日より早いのは，予備時間を取っておくためと課題 2 に食い込むと課題 2 が大変になるため．

・テストとデバッグの日程が長いのは，必要なテスト量が現時点では十分に見積もれないため．

・講義やその他の予定の都合により，個人個人の計画はこのスケジュール通りにはならない．

・(f)と(g)の間は予備日である．

・表内の見積もり時間等は，単なる例であり，いい加減な数値である．各自で見積もること．
・これは表形式で記述されているが，実務では（複数人でのプロジェクトのため）線表を書くことが多い．

・先の作業量が読めないときは，計画が次の通り単純になってしまう．しかし，実際に手を付けると必要時間が見えてくるので，その都度計画をより詳細にするのが望ましい．

簡単な事前作業計画の例（日付は過去の例である）

開始予定日	終了予定日	見積もり時間	作業内容
10/3	10/3	1	(a)スケジュールを立てる
10/4	10/7	3	(b)配布された資料等を読む
10/7	10/10	5	(c)字句解析系（スキャナ）の概略設計
10/10	10/20	10	(d)スキャナの作成(コーディング)
10/20	10/22	3	(e)テストデータと想定テスト結果の準備
10/22	10/28	8	(f)テストとデバッグを行う
10/29	10/30	1	(g)レポート作成
10/31	10/31	−	(h)プログラムとレポートの提出

（１２）スキャナの作り方のヒント

スキャナの作成は，かなり難しそうに思えるが，次の点を踏まえれば，見通しがよくなる．

・次に読み込まれる字句は，先頭の文字で，ほぼ何であるかが決まる．

例えば，一文字読み込んで，それが英字であれば，そこから始まる字句は名前かキーワードになる．また，数字であれば，符号なし整数である．"+"であれば，記号"+"しかあり得ない．他も同様である．分離子があるかもしれないので，それを考慮すると全体の構成は次のようになる．

先頭の文字が分離子であれば，それを読み飛ばす（注釈の時は注釈全体を読み飛ばす）
分離子以外の文字であれば，次のように場合分けする
　英字なら，英数字が続く限り読み込む．それがキーワードのどれかならそのキーワードである
　　どのキーワードとも異なれば，名前である．
　数字なら，数字が続く限り読み込む．それは符号なし整数である．
　・・・・・

・最長一致原則がある

例えば，"abc "と入力があった場合，a だけでも名前であるが，ab も abc 名前である．c の次はスペースなので，最長の字句は abc となり，３文字で一つの字句となる．また，"10abc"と入力があった場合には，10 で一つの字句（符号なし整数）で abc 以降は次の字句となる（10a などは字句ではない）．つまり，字句は次の文字まで確認しないと確定しない場合がある（記号"+"のように確定するものもある）．従って，入力は常に１文字先読みしておくと便利である．即ち，1 文字分の文字バッファを持っていて，それに次の

文字が入っているようにする．以降，この文字バッファを

int cbuf;

として，説明する．

(12-1)初期化関数　int init_scan(char *filename)

filename が表すファイルを入力ファイルとしてオープンする．オープンに失敗したらエラーで終わる．成功したら, そのファイルから 1 文字 cbuf に読んでおく. もし, このとき, EOF が起こったら cbuf には EOF コードを入れる，

(12-2)トークンを一つスキャンする関数　int scan()

switch 文で処理が分かれる

```
switch(cbuf) {
    空白はタブ等の時は，読み飛ばして(cbuf に次の文字を入れて）最初に戻る.
    英字の時は，名前かキーワードである．英数字が続く限り読み込んで(cbuf から適当な文字列バッファへ取り出し，cbuf へ次の文字を読み込むことを繰り返す)，cbuf の内容が英数字以外になったら，そこで英数字列が終わるので，文字列バッファの中身がキーワードかどうか判別して，キーワードならそのトークンコードを，それ以外なら名前のトークンコードを返す.
    数字の時は，数字が続く限り読み込んで，その数字列が表す値を num_attr に入れ，「符号なし整数」のトークンコードを返す.
    注釈の時も字句と同様に注釈の終わりまで読むが，トークンコードを返さずに先頭に戻る
   他のトークンの時も同様である．（以下略）
```

もちろん，switch 文の代わりに，if(...)...else if(...)... を使ってもよいし，これらを組み合わせてもよい.

EOF には，気を付けること．特に，文字列やコメント内の処理中に EOF が現れても無限ループにならないようにすること.

(12-5)行番号関数　int get_linenum()

行番号を数える変数を用意し，cbuf に改行を読み込んだときに，行番号をカウントアップすればよい．ただし，前述のように改行が二つの文字コードの並びである可能性があることに注意せよ．初期化関数でこの変数を初期化しておく．この関数が呼ばれた時にその変数の値を返せばよいように思えるが，スキャナの内部で先読みをする場合があり，先読み文字として newline を読んだときには行番号がずれることがある．それを避けるために，トークンの 1 字めを読んだときに､そのトークン用の行番号を確定するようにするとよい．そして，そのトークンを scan()で返してから次の scan()が呼ばれるまではその番号を返すようにする.