4 字句解析の演習

4.1 演習の概要

- 具体的にすること
  1. mini-C の字句解析を行うためのプログラム `lex.c` を作成する.

     大枠はあらかじめ作成してあるので, それをダウンロードし, 指示に従ってこれにコードを書き込んでプログラムを完成させる.
  2. テストプログラム `testlex.c` とリンクして, 動作確認を行なう.
- 作成する字句解析プログラム
  - `lex.c` は指定されたファイルから mini-C プログラムのトークンを 1 つづつ切り出すための関数群から成る (具体的な関数の仕様は付録参照).
  - mini-C のトークンの種類は次の通り (詳細は付録参照).
    - 識別子
    - キーワード (`int`, `char`, `if`, `else`, `while`, `return` の 6 個)
    - 定数 (整数と文字)
    - 演算子 (`+`, `-`, `*`, `/`, `%`, `&`, `=`, `==`, `!=`, `>`, `>=`, `<`, `<=` の 13 個)
    - 記号 (`,`, `;`, `(`, `)`, {, }, `[`, `]` の 8 個)

4.2 プログラムとデータのダウンロード

1. 講義のホームページ (`http://ist.ksc.kwansei.ac.jp/~ishiura/cpl/`) の「プログラム」を選択する.
2. 次のプログラムとデータをダウンロードする.

`lex.h`	字句解析プログラムのヘッダ
`testlex.c`	テストプログラム
`testlex.txt`	テストデータ
`testlex_out.txt`	テストの期待値 (正解)
`lex.c`	字句解析プログラムのテンプレート*

   - `lex.c` については日本語の文字コードを選択する. 特に事情がなければ UTF-8 を選択すればよい.
   - `lex.c` のダウンロードの時だけブラウザの応答が違うかも知れないが, 気にしなくてよい.
3. 自分の計算機でプログラムがコンパイルできるかどうかテストする.

   `"-c"` と `"-o"` を間違えないように注意.

```
gcc -c lex.c
gcc -c testlex.c
gcc -o testlex testlex.o lex.o
```

   - ここでエラーが出たら助けを呼ぶ.
   - 問題がなければ, 実行可能ファイル `testlex` (Unix 系) か `testlex.exe` (Cygwin) ができている (実行すると, コマンドラインの文法を表示して終わる).

Segmentation fault (セグメンテーション違反) とは

プログラムがアクセスしてはならない主記憶の領域にアクセスした場合にこのようなエラーが出る. 原因はさまざまだが, 初歩的なミスとしては次のようなものが挙げられる.

1. 配列の添字が範囲外だった.

   例えば, `int a[10]` と宣言された配列に対し, `a[-100]` や `a[9200]` にアクセスした. あるいは, `i` が初期化されていない状態で, `a[i]` にアクセスした, 等.

2. 不正なポインタでデータにアクセスした.

   例えば, 初期化していないポインタ `p` で `*p` にアクセスした, 文字列の最後の `'\0'` を忘れたために延々とメモリーアクセスが続いた, 等.

4.3 testlex.c の動作とその中身

1. `testlex.c` は `lex.c` の動作をテストするプログラム

   - このプログラムは, `lex.c` が完成した暁にはそれを使って, 指定されたファイルからトークンを次々に取り出して表示する. 例えば,

```
int main() {
  int year;
  year = year + 1900;
  return 0;
}
```

   が格納されたファイル `test.mc` に対して

```
./testlex test.mc
```

   を実行すると (一番最初の段階ではエラーになるが, プログラムの作成が進めば),

```
type = KW_INT     token = "int"
type = ID         token = "main"
type = LPAR       token = "("
type = RPAR       token = ")"
type = LBRACE     token = "{"
type = KW_INT     token = "int"
type = ID         token = "year"
type = SEMICOLON  token = ";"
type = ID         token = "year"
type = EQ         token = "="
type = ID         token = "year"
type = PLUS       token = "+"
type = INT        token = "1900"  val = 1900
type = SEMICOLON  token = ";"
type = KW_RETURN  token = "return"
type = INT        token = "0"  val = 0
type = SEMICOLON  token = ";"
type = RBRACE     token = "}"
```

が表示されて, 字句解析が行われていることを確認することができる.

2. testlex.c プログラムの中身を見てみよう.

これで lex.c がどういう仕様なのかが大まかに理解できてくるはず.

[testlex.c]

```
 1:  #include <stdio.h>
 2:  #include <stdlib.h>
 3:  #include <string.h>
 4:  #include "lex.h"
 5:
 6:  int main(int argc, char **argv)
 7:  {
 8:     char filename[FILENAME_MAX];
 9:     lex_t *x;
10:
11:     if (argc<2) {
12:        fprintf(stderr, "SYNTAX: %s filename\n", argv[0]);
13:        exit(EXIT_FAILURE);
14:     }
15:     strcpy(filename,argv[1]);
16:
17:     x = lex_new(filename);
18:     lex_get(x);
19:     while(x->type != token_EOF) {
20:        printf("type = %-10s", lex_typename(x->type));
21:        printf(" token = \"%s\"", x->token);
22:        if (x->type==token_INT || x->type==token_CHAR) {
23:           printf("  val = %d", x->val);
24:        }
25:        printf("\n");
26:        lex_get(x);
27:     }
28:     lex_delete(x);
29:
30:     return 0;
31:  }
```

- 4 行目の lex.h は lex.c の変数や関数の宣言をまとめたヘッダファイルである. これを include することによって, lex.c の変数や関数の名前や型をコンパイラが認識できるようになる.
- 6 行目の main の 2 つの引数は, Unix や Cygwin のコマンドラインから ./testlex を起動した時の引数の数 (argc) と引数 (argv) である. 引数は文字列の形で渡され, i 番目の引数は argv[i] である. (argv[i-1 でないことに注意! argv[0] にはこのプログラムを起動したコマンド名が入っている.)
- 8 行目の FILENAME_MAX は, ファイル名の最大長 (正確には最大長のファイル名を格納するのに必要な char 配列の要素数) で, stdio.h で定義されている.
- 9 行目の lex_t は, 字句解析の処理に必要なデータをまとめた構造体であり, その構造体へのポインタ x を宣言している.

– 11〜15 行目はコマンドライン引数の解析. 引数の数が 0 以下 (argc が 1 以下) の場合にはエラーとする. argv[1] には ./testlex test.mc とコマンドを打ち込んだ時の "test.mc" が入っている. これを filename にコピーする.
– 17 行目では, filename を引数としてして lex_new を呼び出している. lex_new は, 引数で渡された名前のファイルをオープンするとともに, 字句解析のために必要な構造体を割り当てて, データの初期設定を行い, その構造体へのポインタを返す.
– 18〜27 行目の while ループは, 次々にトークンを切り出しながらそれを表示する. lex_get は次のトークン一つを切り出す関数であり, 切り出したトークンのタイプ (トークン番号) を x->type にセットする. トークンのタイプが EOF (end of file) なら終了する. lex_typename は切り出されたトークンのタイプを表す文字列, x->token は切り出されたトークンそのもの (文字列) である. x->val には, トークンが整数リテラルの場合にはその値, 文字リテラルの場合にはその文字コードが格納される.
– 28 行目の lex_delete は, ファイルをクローズし, x の指す構造体を削除する.

4.4 演習

4.4.1 注意事項

- 以下の各項目に従ってプログラムを作成せよ.
- 各項目について, レポートに次を簡単に記せ.

  ※ 課題 L.1 〜 課題 L.8 までは, 特に問題がなければ「特に問題なし」だけでよい.

  1. 何を行ったか.
  2. 結果
  3. 感想
  4. (あれば) 工夫した点, 困った点
- 最後に, この演習全体の感想を記せ.
- 1 項目完了する毎にコンパイルと実行をして問題がないか確認してから先に進むこと. まとめてプログラムを入力してコンパイル&実行しようとすると, エラーやバグの場所が非常に分かりにくくなり, ドツボにはまるので注意 (実績あり).
- lex.c のプログラムは後の構文解析の演習で用いるので, 各自保存しておくこと (バックアップを作っておくことが望ましい).
- lex.h は気に入らなくても書き換えないこと.

4.4.2 補助的関数の完成

課題 L.1 lex_err() の完成

lex_err(lex *x, char *message) は, 字句解析に関連する処理中にエラーが発見されたときに呼び出す関数で, 今読んでいるファイル名, 今読んでいる行の行番号, および引数として受け取ったメッセージ message を表示して終了する.

ダウンロードした lex.c 中の/* [課題 L.1] lex_err() の完成 */ という行の下に, 下記の指示にしたがってプログラムを書き込み, 関数を完成させよ.

1. 次の 4 行を書き込めば完了

```
1:  assert(x!=NULL);
2:  assert(message!=NULL);
3:  fprintf(stderr, "%s:%d: %s\n", x->filename, x->linenum, message);
4:  exit(EXIT_FAILURE);
```

- 1〜2 行目では, 引数として渡されるポインタ x と message が NULL でないかどうかをチェックしている. このコードは無くてもプログラムは正しく動く. しかし, 誰 (自分?) がどこでミスするかわからないので, ガードを固めておく.
- 3 行目では 標準出力にエラーメッセージを出力している. x は, 字句解析に関する情報を保持している構造体へのポインタで, x->filename には現在解析中のファイル名を, x->linenum には現在解析中の行番号を保持している.
- 4 行目の exit 文で強制終了している. EXIT_FAILUE は問題が起こった場合の返り値 (gcc では 1 に設定されている). ちなみに, 正常終了の場合には, EXIT_SUCCESS (gcc では 0) を返すことになっている. (main で最後に return 0; と書くのはこのため.)

2. コンパイルしてエラーがないことを確認せよ.

```
gcc -c lex.c
```

課題 L.2 lex_new の完成

lex_new(lex *filename) は, 字句解析に関するデータを保持するための構造体の領域を割当てて初期設定し, そのポインタを返す. 同時に, filename で指定された名前のファイルのオープンも行う. lex.c を使う時には, 必ず始めに lex_new を呼び出し, この構造体へのポインタを取得しなければならない.

前課題と同様に, /* [課題 L.2] lex_new() の完成 */ の下に, 下記の順にプログラムを書き込み, 関数を完成させよ.

1. 領域の割当て

   - まず, lex_t 型構造体の領域を malloc() で割り当てる. 例えば次のように書ける.

```
1:  lex_t* x = (lex_t*) malloc(sizeof(lex_t));
2:  assert(x!=NULL);
```

   - malloc が失敗した場合には NULL が返るので, assert でチェックを行っている. malloc は成功するとは限らないので, 何らかのチェックを行うコードを書くようにする.

2. 初期設定

   - lex_t 型構造体の初期値を設定する.

     lex_t は lex.h で次のように定義されている.

```
typedef struct {
  char  *filename;
  FILE  *fp;
  int  c;
  int  linenum;
  lex_type_t  type;
  char  token[lex_TOKEN_MAXLEN+1];
  int  token_len;
  int  val;
  lex_trace_t  trace;
} lex_t;
```

各メンバーの意味と設定すべき初期値は次の通り.

- filename はファイル名. lex_new の引数の filename をコピーして初期値とする.
- fp はファイルを読むためのファイルポインタ. ファイルをオープンして取得する.
- c は現在読んでいる 1 文字. 初期値は ' '.
- linenum は現在読んでいる行の番号. 初期値は 1.
- type は現在解析中のトークンのタイプで, 初期値は「未定義」を表す token_UNDEF. lex_type_t はトークンのタイプを表す型 (enum 型) で, lex.h 中に定義されている.
- token 現在のトークン (文字列). 初期値は "". lex_TOKEN_MAXLEN はトークンの最大長であり, lex.h 中に定義されている.
- int token_len は現在のトークンの長さ. 初期値は 0.
- val は整数リテラルや文字リテラルの整数値. 初期値は 0.
- trace はトレース (デバッグ用の追跡出力) のモード. lex のトレースのモードには次の 3 つがあり (lex.h で定義), 初期値は lex_TRACE_NO.

  lex_TRACE_NO … 何もしない

  lex_TRACE_BY_CHAR … 文字単位で出力

  lex_TRACE_BY_TOKEN … トークン単位で出力

具体的には, 1. に続いて次のようなコードを書けばよい. (??? に何が入るかは上の文章を読んで埋めよ).

```
 1:  assert(filename!=NULL);
 2:  x->filename = (char*) malloc(sizeof(char)*(strlen(filename)+1));
 3:  assert(x->filename!=NULL);
 4:  strcpy(x->filename,filename);
 5:  if ((x->fp = fopen(x->filename,"r")) == NULL) {
 6:     fprintf(stderr, "lex:  cannot open file %s\n", x->filename);
 7:     exit(EXIT_FAILURE);
 8:  }
 9:  x->c = ???;
10:  x->linenum = ???;
11:  x->type = ???;
12:  token_reset(x);
13:  x->val = ???;
14:  x->trace = lex_TRACE_NO;
```

- 2 行目: filename と同じ長さの文字列を格納するための領域を割り当てている. 文字列長＋ 1 の領域 (文字＋末尾の '\0' の分) が必要なことに注意.
- 5〜8 行目: ファイルを fopen でオープンし, 取得したファイルポインタを x->fp に設定する.
- 12 行目: token と token_len の初期化は, 別途 token_reset を呼び出して行う (後で完成させる).

3. lex_typename のための表の初期化

- 課題 L.5 で完成させる関数 lex_typename で使用する表の初期化を行う. コードとしては, 関数 define_lex_typename を呼び出すだけ.

```
1:  define_lex_typename();
```

4. ポインタを返す
   - 最後に, 割当てた構造体へのポインタを返す.

```
return x;
```

5. コンパイルしてエラーが出ないことを確認せよ.

課題 L.3 lex_delete の完成

lex_delete(lex *x) は, lex 構造体 x の内部領域の解放とファイルのクローズ, x そのものの領域の解放を行なう.

1. コードは次の通り

```
1:  assert(x!=NULL);
2:  fclose(x->fp);
3:  free(x->filename);
4:  x->filename = NULL;
5:  free(x);
6:  x = NULL;
```

   - 2 行目: ファイルのクローズ.
   - 3〜6 行目: malloc で割り当てた x->filename と x の解放. free 後にはそれを誤って使用しないように, 念のために NULL を代入しておく.
   - ちなみに, x->token の領域は, x を malloc したときに自動的に割り当てられ, x を free するときに自動的に解放される.

2. コンパイルしてエラーが出ないことを確認せよ.

課題 L.4 lex_trace_set の完成

lex_trace_set(lex *x, lex_trace_t trace) は, x->trace に値を設定するだけの関数. この関数を呼び出すことにより, デバッグ情報の出力レベルを切り替える.

1. コードは次の通り

```
1:  assert(x!=NULL);
2:  assert(trace==lex_TRACE_NO ||
3:     trace==lex_TRACE_BY_CHAR ||
4:     trace==lex_TRACE_BY_TOKEN);
5:  x->trace = trace;
```

2. コンパイルしてエラーが出ないことを確認せよ.

課題 L.5 lex_typename の完成

lex_typename(lex_type type) は, 字句の種類 type (実際には整数値) を受け取り, その文字列表現を返す. 例えば, トークンの種類がキーワードの else の場合, ソースプログラム中では token_KW_ELSE と書くが, 実際の内部表現は整数値の 6 なので, デバッグ等を行うときにそのまま表示したのでは意味が分かりにくい. そこで, これを "KW_ELSE" 等という文字列に変換するのが関数 lex_typename の役割である.

1. 字句の種類の内部表現と文字列表現との対応は, 関数 define_lex_typename で定義している. ここでは配列 lex_typename_ に各字句の種類に対する文字列を書き込んでいる. 関数 define_lex_typename は, 演算子 (+, -, *, /, %, &) の処理を除いて完成している. 他の部分を参考に, [課題 L.5] の下に演算子の処理を書き込み, この関数を完成させよ.

2. 関数 lex_typename 自体は次の 3 行だけである.

```
1:  assert(token_BEGIN_<=type && type<token_END_);
2:  assert(lex_typename_[type][0]!=0);
3:  return lex_typename_[type];
```

   - 1 行目: 引数として渡された type が範囲内であることの確認. token_BEGIN_ は最初のトークンの番号, token_END_ は最後のトークン番号より 1 大きい値であり, lex.h 内で定義してある.
   - 2 行目: 配列 lex_typename_ が定義されているかどうかの確認.
   - 3 行目: 配列 lex_typename_ の対応する要素を返す.

3. コンパイルしてエラーが出ないことを確認せよ.

課題 L.6 c_get() の完成

c_get(lex *x) は, x->fp で指されるファイルから 1 文字読み込み, それを x->c にセットするというものである.

1. 主要な処理

   1 文字の読み込みには, ファイルポインタで指定されるファイルから 1 文字を読み取って返す関数 fgetc(FILE*) を用いる.

```
1:  assert(x!=NULL);
2:  x->c = fgetc(x->fp);
```

2. 行番号の処理

   読んだ記号が改行記号があれば, 行番号を増やす.

```
3:  if (x->c=='\n') { x->linenum++; }
```

3. トレースの出力

   トレースモードが lex_TRACE_BY_CHAR であれば, 読んだ 1 文字を出力する.

```
4:  if (x->trace==lex_TRACE_BY_CHAR) {
5:      if (x->c==EOF) {
6:          fprintf(stderr, " EOF\n");
7:      }
8:      else {
9:          fprintf(stderr, "%c", (char)(x->c));
10:     }
11: }
```

4. コンパイルしてエラーが出ないことを確認せよ.

課題 L.7 token_reset の完成

token_reset(lex *x) は, トークンを格納する変数 x->token を初期化する (長さ 0 の文字列をセットする).

1. プログラム

   - x->token_len を 0 にし, x->token を "" に初期化する. 具体的なコーディングは各自考えよ.

     ※ 「x->token の初期化」を x->token = ""; とするのは間違い. 正解は脚注 [1] に書いておくが, できれば自力で考えてみること.

[1] 正解は, x->token[0] = '\0'; か strcpy(x->token, "");

2. コンパイルしてエラーが出ないことを確認せよ.

課題 L.8 token_c_append の完成

token_c_append(lex *x) は, トークンを格納する変数 x->token の末尾に読み込んだ文字 x->c をくっつける. ただし, トークンの長さが最大許容量 lex_TOKEN_MAXLEN を越えた場合は, これ以上追加しないようにする.

1. プログラム

   - 文字の追加は, x->token の末尾 (もともと '\0' のあった位置) に x->c を代入してトークンの長さ x->token_len を 1 増やし, (新しい) 末尾に '\0' を代入する.

```
assert(x!=NULL);
if (x->token_len < lex_TOKEN_MAXLEN - 1) {
  x->token[x->token_len] = ???;
  x->token_len++;
  x->token[x->token_len] = '\0';
}
```

2. コンパイルしてエラーが出ないことを確認せよ.

4.4.3 字句解析処理のプログラミング

以下では, 字句解析処理のメインとなる関数 lex_get のプログラミングを行う.

lex_get(lex *x) は次のトークンを 1 つ切り出し, 文字列 x->token に格納する. 同時に, トークンのタイプ (番号) を x->type にセットする. さらに, トークンが整数リテラルの場合にはその整数値としての値, トークンが文字リテラルの場合にはその文字コードを表す整数値を x->val にセットする.

lex_get のプログラムは下記のようになっている.

```
 1:  void lex_get(lex_t *x)
 2:  {
 3:     assert(x!=NULL);
 4:     token_reset(x);
 5:     x->type = token_UNDEF;
 6:     x->val = 0;
 7:
 8:     lex_skip_space(x);
 9:
10:     if (isalpha(x->c)) { lex_get_kw_id(x); }
11:     else if (isdigit(x->c)) { lex_get_int(x); }
12:     else if (x->c=='\'') { lex_get_char(x); }
13:     else { lex_get_others(x); }
14:
15:     if (x->trace==lex_TRACE_BY_TOKEN) {
16:        fprintf(stderr, "(linenum(%d) type(%s) token(\"%s\"))\n",
17:           x->linenum, lex_typename(x->type), x->token);
18:     }
19:  }
```

- 4〜6 行目: トークンとトークンに関連する値の初期化
  - トークンを "" にし, タイプは未定義とし, 値は 0 とする.

– 8 行目: 空白を読み飛ばす.
– 10〜13 行目: 先頭文字 x->c をみてトークン切り出しの処理を選択
  * 先頭文字がアルファベットなら, キーワードか ID (lex_get_kw_id を呼び出す).
  * 先頭文字が数字なら, 整数リテラル (lex_get_int を呼び出す).
  * 先頭文字がシングルクウォート (') なら, 文字リテラル (lex_get_char を呼び出す).
  * それ以外の場合は, lex_get_others を呼び出す.
– 15〜18 行目: トレースモードが lex_TRACE_BY_TOKEN の時には, 行番号, トークンのタイプ, トークンを出力する.

課題 L.9 空白の読みとばし (lex_skip_space の完成)

1. 処理内容
   - スペース, タブ, 改行記号など, プログラム中で空白と見なされる文字をすべて読み飛ばす.
2. プログラム
   - この処理を行う関数 lex_skip_space を完成させる.
   - 今読んでいる文字 x->c が空白文字である間 c_get(x) を繰り返し呼ぶ.

```
while (x->c が空白文字) {
  c_get(x);
}
```

   - 文字が空白文字かどうかの判定には, 標準関数 isspace(char) を用いよ.
     – isspace は, 与えられた文字が空白文字のとき 0 以外, 空白文字でないとき 0 を返す.
     – isspace を用いるために, lex.c の冒頭で <ctype.h> を include している.
3. 動作確認
   - lex.c と testlex.c をコンパイル&リンクする.

```
gcc -c lex.c
gcc -c testlex.c
gcc -o testlex testlex.o lex.o
```

     testlex.c のコンパイルは毎回する必要はない. 2 回目以降は,

```
gcc -o testlex testlex.o lex.c
```

     だけでよい.
   - testlex にテスト用データ testlex.txt を与えて実行する.

```
./testlex testlex.txt
```

     すると,

```
testlex.txt:4:  invalid token
```

     と表示されて終了するはず (違えばここまでのどこかに間違いがある).

     このメッセージは, testlex.txt の 4 行目まで字句解析が進み, そこでエラーが出たことを意味している. testlex.txt の中身は次の通りで, 3 行目までは空白文字だけである. つまり, 3 行目までにある空白の読みとばしには成功し, 4 行目の a を読んだところで解析が失敗 (まだその先の解析プログラムが未完成なので) したということである.

```
 1:
 2:  (空白が 3 個)
 3:  (タブが 3 個)
 4:  a b c abc ABC main x123 p000Ax3
 5:  char else if int return while
 6:  _program u_p0x_A0x_774
 7:  1 2 12 123 0123456 00123456789
 8:  'a' 'b' '0' '\t' '\n' '\'' '\\'
 9:  ;, () {} []
10:  -+*/%&= == != > >= < <=
```

課題 L.10 先頭がアルファベットだった時の処理 (lex_get_kw_id の完成)

1. 処理内容
   - 先頭がアルファベットということは, 識別子かキーワードのいずれかである. ひとまず, これを識別子と考えて処理し, その後に x->token がキーワードのどれかに一致するればキーワードと判定する.
   - 切り出したトークンを x->token に格納する.
2. プログラム
   - この処理を行う lex_get_kw_id を完成させる.
   - まず, 先頭文字 x->c を x->token に入れる.
     - x->token の初期値は "" なので, token_c_append(x) を呼んで x->c を末尾に追加する
     - 次の 1 文字を読み込む (処理が終われば, 必ず次の 1 文字が読み込まれているようにする).

```
token_c_append(x);
c_get(x);
```

   - 次に, 続く文字がアルファベットか数字でなくなるまで, その文字を x->token の末尾に追加していく.

```
while(x->c がアルファベットまたは数字) {
  token_c_append(x);
  c_get(x);
}
```

   - 文字がアルファベットかどうかは isalpha 関数で, 数字かどうかは isdigit 関数で判定できる.
   - 最後に, 切り出したトークン (x->token) がキーワードに一致すればその種類番号を x->type に代入する. そうでなければ識別子の番号 (token_ID) を x->type に代入する.

```
if      (strcmp(x->token,"char"   )==0) { x->type = token_KW_CHAR; }
else if (strcmp(x->token,"else"   )==0) { x->type = token_KW_ELSE; }
else if (strcmp(x->token,"if"     )==0) { x->type = token_KW_IF; }
… 全キーワードについて同じ処理 …
else { x->type = token_ID; }    /* キーワードに一致しなければ識別子 */
```

3. 動作確認
   - 前の課題と同様に testlex.txt を入力として実行する. 次のような出力が得られれば OK.

```
type = ID          token = "a"
type = ID          token = "b"
type = ID          token = "c"
type = ID          token = "abc"
type = ID          token = "ABC"
type = ID          token = "main"
type = ID          token = "x123"
type = ID          token = "p000Ax3"
type = KW_CHAR     token = "char"
type = KW_ELSE     token = "else"
type = KW_IF       token = "if"
type = KW_INT      token = "int"
type = KW_RETURN   token = "return"
type = KW_WHILE    token = "while"
testlex.txt:6: invalid character
```

- 識別子とキーワードが認識されている.
- 最後のエラーメッセージは, testlex.txt の 5 行目まで処理が進み, 6 行目で解析に失敗したことを示す (6 行目の _ を含む識別子が認識できない).
- char, else, if, int, return, while がキーワードとして認識されているか (例えば, token = "char" の時 type = ID ではなく type = KW_CHAR になっているか), 逆に識別子がキーワードとして認識されていないかどうかよく確認せよ. ここで間違っていると, 構文解析の演習で非常に困ることになる.

課題 L.11 アンダーライン (_) を含む識別子の処理 (lex_get_kw_id の改良)

1. 処理内容
   - isalpha(c) では '_' はアルファベットではないと判定される. そこで, c がアルファベットか '_' を判定するマクロ関数 isalpha_(c) を新たに定義する.
2. プログラム
   - 冒頭の /* マクロ関数の定義 */ のところで, isalpha_(c) を定義する.

```
#define isalpha_(c) (isalpha(c) || (c)=='_')
```

   - isalpha を全て isalpha_ に書き換える.
3. 動作確認
   - 前課題と同様に実行すると, 解析が次のように 6 行目まで進む.

```
   ...
type = ID          token = "_program"
type = ID          token = "u_p0x_A0x_774"
testlex.txt:7: invalid token
```

課題 L.12 先頭が数字だった時の処理 (lex_get_int の完成)

1. 処理内容
   - 先頭が数字ということは, このトークンは整数リテラル (token_INT) である.
   - トークンの切り出し方は, 識別子の場合と同様.
   - 整数リテラルの場合には, その値を計算し x->val に格納する必要がある. 例えば, トークンが "123" (文字列) ならば, x->val = 123 (整数値) としなければならない.

2. プログラム
   - この処理を行う lex_get_int を完成させる.
   - まず, このトークンのタイプ (x->type) が整数リテラルであることを記録する.

```
x->type = token_INT;
```

   - 先頭文字を x->token に入れ, 最初の 1 文字の数字としての値を x->val にセットする.
   - x->c の「数字としての値」は x->c - '0' で求められる.

```
token_c_append(x);
x->val = x->c - '0';
c_get(x);
```

   - 次に, 続く文字が数字である限りその文字を x->token に追加していく.
   - x->val はこれまでに得られている値を 10 倍したものに今読んだ文字の値を足せばよい. (以下の ??? の部分は各自考えよ.)

```
while( ???  )  {
  token_c_append(x);
  x->val = ???;
  c_get(x);
}
```

3. 動作確認
   - 実行すると, 7 行目まで処理できる.

```
   …
type = INT        token = "1"  val = 1
type = INT        token = "2"  val = 2
type = INT        token = "12"  val = 12
type = INT        token = "123"  val = 123
type = INT        token = "0123456"  val = 123456
type = INT        token = "00123456789"  val = 123456789
testlex.txt:8: invalid token
```

     – 特に, val の値が正しく計算されているかどうか確認せよ.

課題 L.13 文字リテラルの処理 (lex_get_char の完成)

1. 処理内容
   - 先頭がシングルクォート (') の場合, このトークンは文字リテラル (token_CHAR) である.
   - シングルクォートに囲まれた文字のコードを x->val にセットする.
   - この際, バックスラッシュ (\) のついた文字 ('\n' 等) に注意.
2. プログラム
   - この処理を行う lex_get_char を完成させる.
   - まず, トークンのタイプが文字リテラルであることを記録し, 最初の 1 文字 (') をトークンに入れ, 次の文字を読み込む.

```
x->type = token_CHAR;
token_c_append(x);
c_get(x);
```

- シングルクオートの次の文字のコードを x->val に格納する.
  - 通常の文字 ('a' など) は次の 1 文字だけ見れば良い.
  - しかし, backslash (\) がある時は, 次の 2 文字を見て処理しなければならない.

```
if (x->c=='\\') {
  token_c_append(x);
  c_get(x);
  if      (x->c=='n') { x->val = '\n'; }
  else if (x->c=='\'') { x->val = '\''; }
    … '\t' や '\\' も同様 …
  else { lex_err(x, "invalid character"); }
}
else { x->val = x->c; }
token_c_append(x);
c_get(x);
```

- 最後に, 閉じる方のシングルクォートを確認.
  - なければ, 'abc' などのように文字として長過ぎるわけなので, エラーとする.

```
if (x->c!='\'') { lex_err(x, "character too long"); }
token_c_append(x);
c_get(x);
```

3. 動作確認
   - 実行すると, 8 行目まで処理できる.
     - 文字コードが val にセットされていることを確認せよ.

```
   …
type = CHAR       token = "'0'"  val = 48
type = CHAR       token = "'\t'"  val = 9
type = CHAR       token = "'\n'"  val = 10
type = CHAR       token = "'\''"  val = 39
type = CHAR       token = "'\\'"  val = 92
type = SEMICOLON  token = ";"
type = COMMA      token = ","
testlex.txt:9: invalid character
```

課題 L.14 区切り記号の処理 (lex_get_others の前半)

1. 処理内容
   - 区切り記号は全て 1 文字なので, 読んだ文字が区切り記号なら即座にそのトークン番号を返せばよい.
2. プログラム
   - この処理は lex_get_others の前半で行っている.

- コンマとセミコロン (;) の例に習って, 括弧 ((,)), カーリーブレース ({, }), ブラケット ([,]) の処理を付け加えよ.

```
if (x->c==',') {token_c_append(x); c_get(x); x->type = token_COMMA;}
else if (x->c==';') {token_c_append(x); c_get(x); x->type = token_SEMICOLON;}
 ...
```

3. 動作確認
   - 実行すると, 9 行目まで処理できる.

```
   ...
type = SEMICOLON  token = ";"
type = COMMA      token = ","
type = LPAR       token = "("
type = RPAR       token = ")"
type = LBRACE     token = "{"
type = RBRACE     token = "}"
type = LBRACK     token = "["
type = RBRACK     token = "]"
testlex.txt:10: invalid character
```

課題 L.15 演算記号の処理 (lex_get_others の後半)

1. 処理内容
   - 区切り記号とほぼ同様だが, 2 文字で 1 トークンとなるような記号 (== など) に注意
2. プログラム
   - この処理は lex_get_others の後半で行っている.
   - プラス記号の例に習って, 1 文字の演算記号-, *, /, %, & を完成させよ.

```
else if (x->c=='+') {token_c_append(x); c_get(x); x->type = token_PLUS;}
...
```

   - 最初の文字が =の場合には, == と = の 2 つの可能性がある. このような場合には, 次のように 1 文字先読みして決定する. 他の演算記号 (<, <=, >, >=, !=) も同様.

```
else if (x->c=='=') {
  token_c_append(x);
  c_get(x);
  if (x->c=='=') {
    token_c_append(x);
    c_get(x);
    x->type = token_EQEQ;
  }
  else { x->type = token_EQ; }
}
```

   - lex_get_others の最後の 2 行は, EOF (end of file) の処理と, 処理対象の文字と一致しなかった場合のエラー処理である.

```
else if (x->c==EOF) { x->type = token_EOF; }
else { lex_err(x, "invalid character"); }
```

3. 動作確認

- 実行すると, 最後まで処理できる.

```
   ...
type = MINUS      token = "-"
type = PLUS       token = "+"
type = STAR       token = "*"
type = SLASH      token = "/"
type = PERCENT    token = "%"
type = AND        token = "&"
type = EQ         token = "="
type = EQEQ       token = "=="
type = NE         token = "!="
type = GT         token = ">"
type = GE         token = ">="
type = LT         token = "<"
type = LE         token = "<="
```

  - >= や != がきちんと処理され, 2 つの演算記号に分解されていないかどうか確認すること.

- 念のため, 実行結果をファイル tmp.txt に保存し, 正解と完全に一致しているかどうかを確認せよ.

```
./testlex testlex.txt > tmp.txt
diff tmp.txt testlex_out.txt
```

一致していれば何も出力されない. 不一致があると, 不一致行が出力される.

Nagisa ISHIURA

付録 lex.h/lex.c の仕様

1. マクロ定数

lex_TOKEN_MAXLEN	トークンの最大長 (これ以上の長さのトークンは末尾が切り捨てられる)
token_BEGIN_	lex_type_t 型の最初のトークンの番号
token_END_	lex_type_t 型の最後のトークンの番号+1

2. 型

lex_type_t トークンの種類を表す列挙型

token_UNDEF	未定義
token_EOF	ファイルの終り (end of file)
token_ID	識別子 (変数名, 関数名)
token_INT	整数リテラル (123 など)
token_CHAR	文字リテラル ('c', '\n' など)
token_KW_CHAR	キーワード char
token_KW_ELSE	キーワード else
token_KW_IF	キーワード if
token_KW_INT	キーワード int
token_KW_RETURN	キーワード return
token_KW_WHILE	キーワード while
token_PLUS	演算子 +
token_MINUS	演算子 -
token_STAR	演算子 *
token_SLASH	演算子 /
token_PERCENT	演算子 %
token_AND	演算子 &
token_EQ	演算子 =
token_EQEQ	演算子 ==
token_NE	演算子 !=
token_GT	演算子 >
token_GE	演算子 >=
token_LT	演算子 <
token_LE	演算子 <=
token_COMMA	記号 ,
token_SEMICOLON	記号 ;
token_LPAREN	記号 (
token_RPAREN	記号)
token_LBRACE	記号 {
token_RBRACE	記号 }
token_LBRACK	記号 [
token_RBRACK	記号]

lex_trace_t デバッグ用のトレース情報出力のモードを表す列挙型

lex_TRACE_NO	何もしない
lex_TRACE_BY_CHAR	文字単位で出力
lex_TRACE_BY_TOKEN	トークン単位で出力

lex_t 字句解析用のデータを記憶する構造体

char* filename	処理中のファイル名
FILE *fp	ファイルにアクセスするためのポインタ
int c	現在読んでいる 1 文字
int linenum	現在読んでいる行の番号
lex_type type	直近に切り出したトークンのタイプ
char token[lex_TOKEN_MAXLEN]	トークンそのもの
int token_len	トークン長
int val	整数リテラルや文字リテラルの値
lex_trace_t trace	デバッグ用のトレース情報出力のモード

3. 関数

`lex_t* lex_new(char *filename)`	字句解析のためのデータ (`lex_t` という型の構造体) の割当てと初期化を行う. `filename` というファイルをオープンし, データを読み込めるようにする.
`void lex_get(lex *x)`	次のトークンを 1 つ切り出す. 切り出したトークンは文字列 `x->token` に格納すると同時に, トークンの型を `x->type` にセットする. さらに, トークンが整数リテラルの場合にはその整数としての値を, 文字リテラルの場合にはその文字コードの整数値を `x->val` にセットする.
`void lex_trace_set (lex *x, lex_trace_t trace)`	デバッグ用にトレース (実行状況の出力) を行うモードを設定する.
`char* lex_typename (lex_type_t type)`	与えられたトークンの型に対応するトークン種類名を返す. (主として表示用に用いる.)
`void lex_err(lex *x, char *msg)`	エラーメッセージ `msg` を, 現在読み込み中の行の番号や最後に読んだトークンとともに表示し, プログラムを強制終了する.
`void lex_delete(lex *x)`	`lex_new` で割り当てたデータを削除する.

4. lex.c 内部だけで用いるマクロ, 変数, 関数

`lex_TYPENAME_MAXLEN`	トークンの型を表す文字列の最大長
`char lex_typename_ [token_MAX_][lex_TYPENAME_MAXLEN+1]`	トークンの型を表す文字列を格納する配列
`isalpha_(char c)`	文字 `c` が英字か '_' の時 0 以外, そうでないとき 0 を返す.
`void define_lex_typename()`	トークンの型を表す文字列を配列 `lex_typanane_` を初期化する.
`void c_get(lex *x)`	次の一文字を読み込み, `x->c` にセットする.
`void token_reset(lex *x)`	トークン `x->token` を空にする.
`void token_c_append(lex *x)`	トークン `x->token` の末尾に文字 `x->c` を追加する.