

7. Lex による字句解析

- ♣ Mini-C 言語の字句解析系を作成する。C/C++ でプログラミングするのではなく、lex を用いて字句解析系を自動生成する。

7.1 Yacc と Lex

Lex は、正規表現による字句の定義から字句解析系を自動生成するツールである。また yacc は文法の記述から構文解析系を自動生成するツールである。Yacc と lex の組み合わせにより言語処理系を効率的に作成する方法は、プログラミング言語の処理系だけでなく、Unix の諸定義ファイルの読み込み等に広範に用いられており、Unix 上での開発を行う上で不可欠なツールである¹。

7.2 準備と lex の概要

1. 講義ホームページより次のファイルをダウンロードせよ。

- lex.ll … lex ファイルのテンプレート
- lex.h … テスト用プログラムのヘッダ
- testlex.cpp … テスト用プログラム
- testlex.txt … テストデータ (1)
- testlex_out.txt … テストデータ (1) に対する期待値
- factor.mc … テスト用データ (2)

2. lex.ll を見てみる。

lex のファイルは、大まかに 4 つの部分 (ここでは便宜上 A パート～D パートと呼ぶ) からなる。

- A パート (先頭の %{ の行から次の %} の行まで)
ここに書かれたものは、lex が自動生成する字句解析 C プログラムの冒頭にそのまま取り込まれる。ヘッダのインクルードやグローバル変数の宣言等をここに書く。最初の時点では、
 - テストプログラム testlex.cpp とのリンクに必要な lex.h のインクルード
 - 解析中の行番号を格納する整数変数 linenumを宣言している。
- B パート (%} の行から最初の %% の行まで)
字句の正規表現による定義は次の C パートに書くが、複雑な正規表現はここであらかじめ定義する。(現状では空である。後の演習で書き加える。)
- C パート (最初の %% の行から次の %% の行まで)
字句のパターンと、それが見つかったときのアクションを書く。1 行に 1 つの字句の定義を書く。行頭に字句のパターンを正規表現で書き、それに続く { } 内にアクションを C 言語で書く。
現状では、サンプルとしていくつかの字句定義を書いてあるが、例えば、

```
"char" { return lex_KW_CHAR; }
```

¹将来的には XML がこれに変わるのでは、との見方もあるが。

は, char という文字列が見つかったらそれを一つの字句として切り出し, 「キーワードの char」を表すトークン番号 lex_KW_CHAR を構文解析プログラムに返す, ということを意味する.

基本的に, このようなパターンを列挙していくだけで, 字句解析プログラムを自動生成することができる. ただし, トークンのマッチングはここに書かれた順に試されるので, 字句定義の順序に留意する必要がある.

最後の行の

```
. { fprintf(stderr, "Invalid character '%c'\n", yytext[0]); exit(4); }
```

は, いずれにもマッチしなかった場合の処理を書いている. 先頭の “.” (ピリオド) は, 特殊文字で, 「任意の文字にマッチする文字」を意味する. 従って, 解析中の文字列が, C パートに書かれた字句定義のいずれにもマッチしなければ, この部分が実行される. アクションは, エラーメッセージを表示して強制終了するというものである.

- D パート (最後まで)

ここに書かれたものは, 字句解析プログラムの末尾にそのまま取り込まれる. 字句解析に必要な関数の本体等をここに書く. ここでは,

- エラー処理用の関数 yyerror()
- 複数ファイルを処理する場合の関数 yywrap()

を宣言している. 本演習では, この部分の変更は行わない.

3. コンパイル

```
g++ -g -c testlex.cpp          ... これは最初の 1 回だけでよい
flex lex.ll                    ... lex ファイルのコンパイル
g++ -g -o testlex testlex.o lex.yy.c
```

- testlex.cpp はテスト用のプログラムである. 生成された字句解析プログラムの関数を呼び出して, 指定されたファイル中の字句を次々に切り出して表示する.
- flex は GNU 版の lex である. lex.ll を flex によりコンパイルすると, 字句解析を行う C プログラムが lex.yy.c というファイルに出力される. これを testlex.o とリンクして, 動作をテストする. コンパイルには gcc ではなく, g++ を用いよ.

4. 実行

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

- 次のようなファイル testlex.txt を入力として字句解析を行い, トークンを 1 つずつ出力する.

```
( 改行のみ )
( スペースが 2 個 )
( タブが 1 個 )
int char else if return while
;, ( ) { } []
-+*/%&= == != > >= < <=
1 2 12 123 0123456 00123456789
a b c abc ABC main x123 p000Ax3
_program u_p0x_A0x_774
'a' 'b' '0' '\t' '\n' '\'' '\\'
```

- 実行すると, 次の様なエラーで止まる.

```
1: 不正な文字 , ,
```

これは、空白文字の読み飛ばしを定義していないので、1 行目で解析が止まってしまったためである。

7.3 Lex の演習

☆ 各課題にプログラムを添付する必要はない。完成した lex.ll のファイルを一番最後に添付せよ。

課題 7.1 空白の処理

1. C パートの冒頭に次の (1)(2) を書き加える。

```
%%  
  
[ \t\r] {}          ... (1)  
"\n" {linenum++;}  ... (2)  
  
"char" { return lex_KW_CHAR; }  
"int" { return lex_KW_INT; }
```

- (1) 「スペース文字かタブ記号か復帰記号があったら何もしない (読み飛ばす)」ことを表す。
- (2) 「改行文字があったら、行番号を 1 つ増やす」ことを表す。

2. テスト

コンパイルし、

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

を実行せよ。空白を読み飛ばし、4 行目

```
int char else if return while
```

まで処理が進む。int と char は字句定義があるが、次の else は未定義なので、

```
4: type = KW_INT      token = "int"  
4: type = KW_CHAR     token = "char"  
4: 不正な文字 'e'
```

とエラーで止まる。

課題 7.2 キーワードの定義

1. char, int に続き、他のキーワード else, if, return, while の定義を追加せよ。

2. テスト

実行して、

```
4: type = KW_INT      token = "int"  
4: type = KW_CHAR     token = "char"  
4: type = KW_ELSE     token = "else"  
4: type = KW_IF       token = "if"  
4: type = KW_RETURN   token = "return"  
4: type = KW_WHILE    token = "while"  
5: 不正な文字 ';' ;'
```

となることを確認せよ。5 行目の

```
;, () {} []
```

まで処理が進んだが、区切り記号の定義がまだなのでエラーで止まっている。

課題 7.3 区切り記号・演算記号の定義

1. `&`, `==` に習って、他の記号の定義を追加せよ。

基本的に、

```
記号 { return lex_????; }
```

を列挙していただくので、単純作業。

Mini-C で使う記号とニモニック (`lex_????`) の対応は、末尾の付録を参照せよ。

2. テスト

実行して、

```
...
6: type = GE      token = ">="
6: type = LT      token = "<"
6: type = LE      token = "<="
7: 不正な文字 '1'
```

と、6 行目まで解析が完了することを確認せよ。7 行目の

```
1 2 12 123 0123456 00123456789
```

の整数定数が未定義なので、止まっている。

課題 7.4 整数定数

1. B パートに次の 1 行を追加せよ。

```
...
%}

num [0-9]+      ← この 1 行を追加

%{
...
```

- `[0-9]` は 0 か 1 か 2 か … 9, すなわち「数字 1 文字」を表す。+ は 1 回以上の繰り返しを表す。つまり、`num` は「数字の 1 回以上の繰り返し」であることを定義している。

2. C パートに次の 1 行を追加せよ。

```
{num} {sscanf(yytext, "%d", &yylval.val); return lex_INT;}
```

- B パートで定義したパターンは `{ }` をつけて参照する。つまり `{num}` は、数字の 1 回以上の繰り返しである。
- アクション部の `sscanf` では、文字列として得られるトークンを整数値に変換している。`yytext` がトークン (文字列) である。これを整数値に変換 (`%d`) し、`yylval` という共用体の整数メンバ `val` に格納している。

※ `yylval` の定義は、`lex.h` にある。型 (`YYSTYPE` として参照する) は、

```
typedef union {
    int val;
    char* string;
} YYSTYPE;
```

と定義されていて、`val` には整数値を、`string` には文字列データを格納できる。

3. テスト

実行して、

```
...
7: type = INT      token = "1"   val = 1
7: type = INT      token = "2"   val = 2
7: type = INT      token = "12"  val = 12
7: type = INT      token = "123" val = 123
7: type = INT      token = "0123456" val = 123456
7: type = INT      token = "00123456789" val = 123456789
8: 不正な文字 'a'
```

となることを確認せよ。7 行目の整数定数の解析が完了し、8 行目の

```
a b c abc ABC main x123 p000Ax3
```

で止まっている。

課題 7.5 識別子

1. B パートに次の 1 行を追加せよ。

```
id    [a-zA-Z_][a-zA-Z_0-9]*
```

- `[a-zA-Z_]` はアルファベットかアンダーライン (`_`) 1 文字を表す。 `[a-zA-Z_0-9]` はそれに数字を加えたものである。これにより識別子のパターン `id` を定義している。

2. C パートに次の 1 行を追加せよ。

```
{id} { yylval.string = strdup(yytext); return lex_ID; }
```

- 識別子のパターンが見つかった場合には、`yylval` の文字列メンバである `string` にトークン (`yytext`) をセットして、`lex_ID` を返す。

3. テスト

実行して、

```

...
8: type = ID      token = "a"
8: type = ID      token = "b"
8: type = ID      token = "c"
8: type = ID      token = "abc"
8: type = ID      token = "ABC"
8: type = ID      token = "main"
8: type = ID      token = "x123"
8: type = ID      token = "p000Ax3"
9: type = ID      token = "_program"
9: type = ID      token = "u_p0x_A0x_774"
10: 不正な文字 '''

```

となることを確認せよ。9 行目の解析が完了し、10 行目の

```
'a' 'b' '0' '\t' '\n' '\\'' '\\'
```

で止まっている。

課題 7.6 文字定数

1. C パートに次の 1 行を追加せよ。

```
"".""" { yylval.val = yytext[1]; return lex_CHAR; }
```

- 少し読みにくいが、「single quote (') に挟まれた任意の 1 文字」が目的のパターンである。
※ 先にも述べた通り、ピリオド (.) は任意の文字にマッチする特殊文字である。
- `yylval.val` にその文字のコードをセットしている。例えば、`'a'` という文字定数なら、`yytext` は

	0	1	2	3
yytext	'	a	'	\0

のようになっているので、`yytext[1]` がその文字のコードになる。

2. テスト

実行して、

```

...
10: type = CHAR    token = "'a'"  val = 97
10: type = CHAR    token = "'b'"  val = 98
10: type = CHAR    token = "'0'"  val = 48
10: 不正な文字 '''

```

となることを確認せよ。`'a'`、`'b'`、`'0'` の解析は成功したが、`'\t'`、`'\n'`、`'\''`、`'\\'` の解析ができていない。

課題 7.7 エスケープ記号 (\) を含む文字定数

1. `'\t'`、`'\n'`、`'\''`、`'\\'` が解析できるようにせよ。(単純に 4 つのパターンを列挙すればよい。)
2. テスト

下記のように解析が正常に終了することを確認せよ。特に `val` の値が一致していることを確認せよ。

```

...
10: type = CHAR      token = "'a'"  val = 97
10: type = CHAR      token = "'b'"  val = 98
10: type = CHAR      token = "'0'"  val = 48
10: type = CHAR      token = "'\t'" val = 9
10: type = CHAR      token = "'\n'" val = 10
10: type = CHAR      token = "'\''" val = 39
10: type = CHAR      token = "'\\'" val = 92

```

- 2) 期待値と diff を取り, 間違いないことを確認せよ. (ここできちんと確認しておかないと, 構文解析で
ドツボにはまるので要注意.)

```

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

```

何も表示されなければ全て期待値と一致している. 間違っている場合には差分が表示されるので, lex.ll
を修正する.

- 3) 最後に, factor.mc に対して字句解析を実行し, 正常に終了することを確認せよ.

```

./testlex factor.mc

```



Nagisa ISHIURA

付録 Mini-C 言語の字句一覧

ニモニック	トークン
lex_ID	識別子 (変数名, 関数名など)
lex_INT	整数リテラル (123 など)
lex_CHAR	文字リテラル ('c', '\n' など)
lex_KW_CHAR	キーワード <code>char</code>
lex_KW_ELSE	キーワード <code>else</code>
lex_KW_IF	キーワード <code>if</code>
lex_KW_INT	キーワード <code>int</code>
lex_KW_RETURN	キーワード <code>return</code>
lex_KW_WHILE	キーワード <code>while</code>
lex_PLUS	演算子 <code>+</code>
lex_MINUS	演算子 <code>-</code>
lex_STAR	演算子 <code>*</code>
lex_SLASH	演算子 <code>/</code>
lex_PERCENT	演算子 <code>%</code>
lex_AND	演算子 <code>&</code>
lex_EQ	演算子 <code>=</code>
lex_EQEQ	演算子 <code>==</code>
lex_NE	演算子 <code>!=</code>
lex_GT	演算子 <code>></code>
lex_GE	演算子 <code>>=</code>
lex_LT	演算子 <code><</code>
lex_LE	演算子 <code><=</code>
lex_COMMA	記号 <code>,</code>
lex_SEMICOLON	記号 <code>;</code>
lex_LPAREN	記号 <code>(</code>
lex_RPAREN	記号 <code>)</code>
lex_LBRACE	記号 <code>{</code>
lex_RBRACE	記号 <code>}</code>
lex_LBRACK	記号 <code>[</code>
lex_RBRACK	記号 <code>]</code>