Flex - generator analizatorów leksykalnych

Spis treści:

Wprowadzenie

Flex jest generatorem analizatorów leksykalnych (skanerów). Usiłując opisać flex-a w kilku zdaniach można powiedzieć, że:

- flex przyjmuje na wejściu opis w formie wyrażeń regularnych,
- flex konstruuje na podstawie podanych wyrażeń regularnych automat skończony,
- flex produkuje kod skanera w języku C (lub C++),
- wygenerowny przez flexa program pozwala wykrywać w strumieniu wejściowym ciągi znaków zgodne ze wzorcami określonymi przez wyrażenia regularne i wykonywać zadane akcje po rozpoznaniu wzorca.

Działanie flex-a jest schematycznie przedstawione na rysunku:

Wyrażenia regularne

Poniżej zaprezentowany jest zbiór wyrażeń regularnych udostępniach przez flex-a.

Wyrażenie	Obejmuje	Przykład
c	dowolny znak nie będący operatorem	`a`
\c	zwykłe znaczenia znaku	`\*`
"s"	ciąg znaków	`"**"`
.	dowolny znak z wyjątkiem znaku nowej linii	`a.b`
^	początej linii	`^początek`
$	koniec linii	`koniec$`
[s]	dowolny znak ze zbioru	`[0123abc]`
[^s]	dowolny znak nie będący w zbiorze	`[^0123abc]`
$[s_1\text{-}s_2]$	dowolny znak ze zbioru s_1-s_2	`[0-9]`
r*	zero badź więcej wystąpień wyrażenia r	`a*`
r+	jedno badź więcej wystąpień wyrażenia r	`a+`
r?	zero bądź jedno wystąpienie wyrażenia r	`a?`
r{n,m}	od n do m wystąpień wyrażenia r	`a{1,5}`
r{m}	dokładnie m wystąpień wyrażenia r	`a{5}`
r_1r_2	wyrażenie r_1 a następnie r_2	`ab`

$r_1 \mid r_2$	wyrażenie r_1 lub r_2	`a|b`
(r)	wyrażenie r	`(a|b)`
r_1 / r_2	wyrażenie r_1 gdy następuje po nim r_2	`a/b`
<x> r	wyrażenie r pod warunkiem przebywania w stanie x	`<x>abc`

Przykłady:

Wyrażenie regularne	**Akceptowane ciągi znaków**
flex	`flex`
[0-9][^0-9]	`2a, 3%, 5m`
L(R[0-9]|L[0-9])	`LR1, LL0`
"-"?1	`-1, 1`
0x[0-9A-F]+	`0xFF`
DD" "[0-9]{7}	`DD 3452378`

Schemat specyfikacji

Specyfikacja (wejście) dla flex-a to najczęściej plik z rozszerzeniem .l, np. example.l. Specyfikacja ma następujący układ:

```
definicje pomocnicze
%%
reguły
%%
podprogramy pomocnicze
```

Sekcja **definicje pomocnicze** umożliwiaja zdefinowanie pomocniczych nazw dla złożonych wyrażeń regularnych oraz podanie kodu, który będzie bezpośrdnio kopiowany na początek kodu skanera.

Sekcja **reguły** zawiera wyrażenia regularnym wraz z przypisanymi im akcjami w języku C (lub C++).

Sekcja **podprogramy pomocnicze** umożliwia podanie kodu, który ma być skopiowany bezpośrednio na koniec pliku skanera.

Prosty przykład

Poniżej przedstawiony jest specyfikacja flexa dla skanera, który wyszukuje ciąg znaków *lex* i zamienia go na ciąg znaków *flex*.

```
%{
/* Program wyszukuje ciag znakow "lex" i zamienia go na ciag znakow "flex" */
%}

%%

lex     printf("flex");

%%

main() {
  printf("Zamiana lex na flex:\n");
  yylex();
  return 0;
 }
```

Źródła: zamien.l

Uruchomienie

Aby wygenerować kod skanera wpisujemy:

```
flex zamien.l
```

Otrzymany w ten sposób kod należy skompilować:

```
gcc lex.yy.c -o zamien -lfl
```

Skaner można uruchomić z przykladowym plikiem testowym test_zamien:

```
./zamien <test_zamien
```

Makefile

Przykładowy Makefile:

```
PROG = example

all : ${PROG}

lex.yy.c: ${PROG}.l
                flex ${PROG}.l

${PROG}: lex.yy.c
                gcc -o ${PROG} lex.yy.c -lfl
```

Źródła: Makefile

Więcej informacji:
http://developers.sun.com/solaris/articles/make_utility.html
http://www.gnu.org/software/make/manual/html_chapter/make.html#SEC_Top

Rozstrzyganie niejednoznaczności

Gdy więcej niż jedna reguła pasuje do ciągu wejściowego flex stosuje kolejno dwie zasady:

1. Preferowany jest możliwy do uzgodnienia lexem o największej długości
2. Wśród reguł, które uzgadniają lexem o tej samej długości preferowana jest reguła, która występuje wcześniej (innymi słowy przy równej długości ciągów znaków zadziała reguła wcześniejsza w specyfikacji)

```
%%
begin    printf("%s slowo kluczowe\n", yytext);    // regula 1
[a-z]+   printf("%s identyfikator\n", yytext);    // regula 2
.|\n     /* empty */
```

Jeżeli na wejściu pojawi się ciąg znaków `beginner`, to zadziała reguła 2 i flex potraktuje ciąg znaków jako identyfikator, bo reguła `begin` dopasuje 5 znaków, a reguła `[a-z]+` dopasuje 8 znaków.

Jeżeli na wejściu pojawi się ciąg znaków `begin`, to zadziała reguła 1 i flex potraktuje go jako słowo kluczowe, bo obie reguły dopasowują po 5 znaków.

Źródła: begin.l

Rozpoznawanie kontekstu

Zdarza się, że w różnych momentach analizy dla tego samego wzorca mają być

zastosowane różne akcje. Taka sytuacja wymaga rozpoznawania kontekstu przez skaner.

Flex oferuje następujące metody rozpoznawania kontekstu:

- użycie operatorów kontekstu ^, $ i /

```
^ - początek linii,
$ - koniec linii,
/ - prawy kontekst.
```

```
%{
/* passwd.l
   Analiza zawartosci pliku passwd */
%}

%%

^ewa            printf("uzytkownik ewa ma konto w systemie\n");
:$              printf("uzytkownik nie ma zdefiniowanego shell-a\n");
.|\n            ;
```

Źródła: passwd.l | test_passwd

```
%{
/* shutdown.l
   dopasowuje ciag shut, jezeli wystapi po nim ciag down */
%}

%%

shut/down       printf("%s\n", yytext);
.|\n            /* empty */
```

Źródła: shudown.l

Stany

Do rozpoznawania lewego kontekstu można wykorzystać również stany startowe. Zilustrowano to poniżej.

Przykład użycia stanów startowych

```
%{
  /* jaguar1.l
   Rozpoznawanie znaczenia slowa jaguar w zaleznosci od lewego kontekstu
   przy uzyciu stanow */
%}

%s A B

%%
^a              BEGIN(A);
^b              BEGIN(B);
<A>jaguar       printf("Duzy kot\n");
<B>jaguar       printf("Sportowy samochod\n");
.|\n            ;
```

Źródła: jaguar1.l

Ten sam przykład z użyciem flagi

```
%{
/* jaguar2.l
   Rozpoznawanie znaczenia slowa jaguar w zaleznosci od kontekstu
   z wykorzystaniem flagi */
char flag;
%}

%%
^a      flag = 'a';
^b      flag = 'b';

jaguar  switch(flag){
           case 'a':  printf("Duzy kot\n");
                      break;
           case 'b':  printf("Samochod sportowy\n");
                      break;
        }
.|\n    ;
```

Źródła: jaguar2.l

Flex udostępnia również stany wyłączne. W wyłącznym stanie startowym nie działają żadne reguły poza posiadającymi ten stan w prefixie.

```
%{
/* exclusive.l
   demonstruje dzialanie wylacznych stanow startowych */
%}

%x LITERAL
%%

raz                ECHO;
dwa                ECHO;
#                  BEGIN(LITERAL);
<LITERAL>#         BEGIN(INITIAL);
.                  /* empty */
```

Źródła: exclusive.l | test_exclusive

Predefiniowane zmienne i funkcje

Obiekt predefiniowany	Znaczenie
`ECHO`	przekopiowuje dopasowany ciąg znaków na wyjście
`yytext`	dopasowany ciąg znaków
`yyleng`	długość dopasowanego ciągu znaków
`yywrap()`	zwraca 1, gdy jest jeszcze coś na wejściu do przetworzenia; w przeciwnym razie zwraca 0
`yymore()`	powoduje, że następny rozpoznany ciąg zostanie dopisany do aktualnego, w `yytext` otrzymamy ich konkatenację
`yyless(n)`	pozostawia w `yytext` tylko *n* początkowych znaków, pozostałe zwraca z powrotem na wejście
`REJECT`	oznacza: "przejdź do innego wariantu"; porzuca aktualne dopasowanie i powoduje przejście do alternatywnej reguły; aktualnie dopasowany ciąg zostaje zwrócony na wejście
`yyin`	wejście
`yyout`	wyjście
`input`	pobiera następny znak z wejścia
`unput(c)`	wypisuje znak c na wyjście
`yylval`	zmienna globalna wykorzystywana do przekazania atrybutu tokenu przekazywanego do parsera

Przykłady:

```
%{
/* meta-dane.l
   demonstruje dzialanie funkcji yymore */
%}

%%

meta-    ECHO;  yymore();
dane     ECHO;
```

Źródła: meta-dane.l

```
%{
/* very_easy.l
   demonstruje wykorzystanie dyrektywy REJECT */
%}

%%

very      printf("%s ",yytext);  REJECT;
[a-z]+    ECHO;
```

Źródła: very_easy.l

REJECT = yyless(0);

```
%{
/*  block.l
    Zliczanie wystapien ciagow znakow block i lock  */
int block_c, lock_c;
%}

%%

block {
         block_c++;
         /* Po dopasowaniu ciagu block wycofaj
            wszystkie znaki poza pierwszym do ponownej analizy */
         yyless(1);
      }
lock     lock_c++;
\n|.     /* empty */

%%

main(){

   yyout = fopen("scanner_output", "a+");
   yylex();
   fprintf(yyout, "block - %d, lock - %d.\n",  block_c, lock_c);
   fclose(yyout);
}
```

Źródła: block.l | test_block

```
%{
/* pary.l
   Tworzy statystyke wystapien wszystkich par malych liter */

int digram [26][26];
int i, j;
```

```
%}

%%
[a-z][a-z]       {
                   digram[yytext[0]-97][yytext[1]-97]++;
                   REJECT;
                 }
.|\n             /* empty */
%%

main(){

/* wyzeruj tablice */
  for(i=0; i<26; i++)
    for(j=0; j<26; j++)
       digram[i][j] = 0;

  yylex();

/* wypisz tablice */
/* Drukowanie podsumowań często umieszcza się także w funkcji yywrap */

    for(i=0; i<26; i++)
       for(j=0; j<26; j++)
          if (digram[i][j] != 0)
             printf("%c,%c: %d\n", i+97, j+97, digram[i][j]);
}
```

Źródła: pary.l

Inne materiały

Lex - A Lexical Analyzer Generator, M. E. Lesk,
Flex - manual po polsku,
Flex, version 2.5 - A fast scanner generator, Vern Paxson.