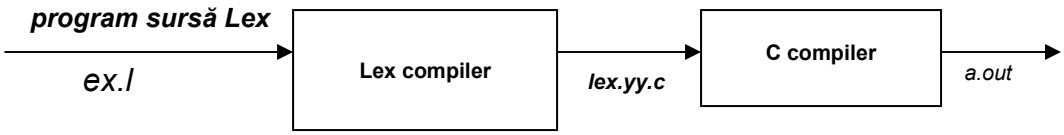
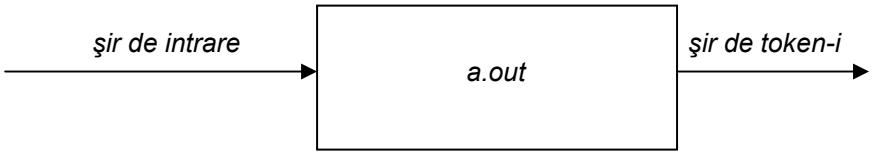
Tehnici de compilare
Laborator 5 - 6

Limbajul LEX (FLEX)

Limbajul LEX a fost introdus de Lesk şi Schmidt în 1975, ca un limbaj pentru specificarea analizorului lexical.

În general Lex-ul este utilizat pentru a genera analizorul lexical, ***a.out***, în modul următor:





Specificații Lex (Flex)

Un program Lex are trei părți:

```
declarații
% %
reguli de traducere
% %
proceduri auxiliare
```

Secțiunea de declarații include declarații de variabile, constante şi definiții regulate. Declarațiile de variabile şi constante şi eventual alte instrucțiuni se scriu în limbaj C şi trebuie incluse între semnele %{ şi %}.

Definițiile regulate sunt similare cu cele prezentate înainte şi sunt folosite drept componente ale expresiilor regulate care apar în regulile de traducere.

Regulile de traducere ale unui program Lex sunt instrucțiuni de forma:

```
p1 {acțiune 1}
p2 {acțiune 2}
..............
pn {acțiune n}
```

unde fiecare p_i este o expresie regulată şi fiecare acțiune *i* este un fragment de program care descrie ce acțiune trebuie executată când forma p_i produce o lexemă. În Lex , acțiunile sunt scrise în limbajul C.

În secțiunea a III-a apar procedurile auxiliare care sunt necesare acțiunilor. Aceste proceduri pot fi compilate separat şi adăugate apoi la analizorul lexical.

Exemplu 1:

Un program LEX simplu, care transformă literele şirurilor de caracter introduse în litere mari.

```
%%
[a-z]           {putchar(toupper(yytext[0]));}
%%
main()
{
     yylex();
}
```

Compilări şi execuție:

Dacă numele programului de mai sus este `ex1.l` atunci trebuie să executăm următoarele comenzi:

```
flex ex1.l
```

Se generează fişierul `lex.yy.c`.

```
gcc -o ex1 lex.yy.c –lfl
```

Compilăm fişierul `C` generat, `lex.yy.c`, generăm executabilul cu numele `ex1` cu opțiunea `-o` iar cu opțiunea`-lfl` realizăm link cu anumite librării.

Executăm fişierul executabil generat:

```
./ex1
```

sau

```
./ex1<in.txt
```

Reguli Lex

O regulă Lex este formată dintr-o expresie regulată şi o acțiune asociată, formată din instrucțiuni C. În momentul în care o expresia regulată este recunoscută pe parcursul citirii caracterelor de către scanner, se execută acțiuniile corespunzătoare se execută.

În specificarea expresiilor regulate Lex suportă o serie de convenții, notații (clase de caractere, repetări specifice, etc.).

Exemplu

```
%{
     static unsigned nrNum = 0;
%}
cifra     0|1|2|3|4|5|6|7|8|9
numar     {cifra}+
%%
{numar}   nrNum++; /*număr câte numere naturale apar în text*/
```

Dacă pentru un *pattern* am mai multe instrucțiuni ce reprezintă acțiunea, acestea se plasează într-un bloc de instrucțiuni, între { }.

Observații:

- Tot ce este scris în prima secțiune a declarațiilor între semnele %{ şi %} şi tot ce se află în ultima secțiune este copiat identic în programul C rezultat din sursa Lex.
- Secțiunile 1 şi 3 sunt opționale
- Regulile din secțiunea 2 trebuie să respecte:
    - încep din prima coloană (nu se lasă spații albe sau tab-uri) în care se scrie *pattern*-ul, dat de o expresie regulată
    - expresiile regulate conțin caractere text (din alfabetul de intrare) şi caractere operator
    - *pattern*-urile sunt separate de acțiuni prin spații albe sau tab-uri.

Expresii regulate în LEX

Expresiile regulate sunt formate din *caractere text* şi din *caractere operator*.
Caracterele text: literele alfabetului şi cifrele.
Caracterele operator: „ \ [ ] ^ - ? . * + | ( ) $ / { } % < >

Observație:

Caracterele operator pot fi transformate în caractere text, dacă sunt precedate de caracterul \ sau prin punerea între ghilimele.

Notații în LEX

Clase de caractere:

Sunt specificate cu ajutorul parantezelor pătrate **[ ]**.

Exemple: **[abc]** reprezintă a sau b sau c
[0-9] înseamnă **0**|**1**|**2**|**3**|**4**|**5**|**6**|**7**|**8**|**9**

[0-9A-Za-z] înseamnă orice literă sau cifră

În afară de caracterele ****, **-** şi **^**, majoritatea celorlalte caractere operator îşi pierd semnificația şi rămân simple caractere text.

Caracterul – (minus): se foloseşte pentru stabilirea unui domeniu:

[0-9] = cifrele de la 0 la 9

Dacă dorim utilizarea caracterului – în interiorul domeniului cu semnificația de caracter text atunci trebuie introdus chiar la început sau precedat de caracterul \.

Caracterul **** are aceeaşi semnificație de operator descris mai sus.

Caracterele operator:

Caracterul **^**	utilizat la începutul unei clase de caractere indică complementara mulțimii specificate. De exemplu, **[^0-9]** înseamnă orice caracter diferit de cifră. **^x** recunoaşte **x** doar la început de linie.
Caracterul **.**	înseamnă orice caracter cu excepția caracterului \n (newline)
Caracterul **$**	**x$** recunoaşte **x** doar la sfârşit de linie.
Caracterul **?**	reprezintă opționalitate: **ab?c** înseamnă şirul abc sau ac.
Caracterul *****	reprezintă stelarea (închiderea Kleene) ca la expresii regulate.
Caracterul **+**	una sau mai multe instanțe (ca la expresii regulate). $\mathbf{a}^{+}$ este echivelent cu **aa***.
Caracterul **|**	reprezintă „sau” logic.
Parantezele **( )**	se folosesc pentru grupare: exemplu: (ab|cd)?(ef)*
Parantezele **{ }**	pot fi folosite pentru desemnarea unui nume din definițiile regulate **{numar}** sau astfel: **a{3}** reprezintă şirul **aaa** (de 3 ori a) **x{3,7}** reprezintă **x** între **3** şi **7** ori. **[a-z]{1,5}** reprezintă orice şir format din litere mici de lungime **1** până la **5**

Operatorul **/** se foloseşte pentru indicarea contextului drept: astfel **ab/cd** semnifică faptul că **ab** este acceptat doar dacă urmează **cd**

Variabile globale Lex

Funcția de citire a tokenilor **yylex** nu are nici un argument şi returnează o valoare întreagă. Alte variabile globale care oferă mai multe informații despre tokenii citiți sunt:

- **yytext** este un şir de caractere null-terminat care conține caracterele lexemei recunoscute. Această variabilă este declarată în fişierul **lex.yy.c**.
- **yyleng** este un întreg care memorează lungimea lexemei memorate în **yytext**. Această variabilă este declarată de asemenea în fişierul **lex.yy.c**.
- **yylval** este o variabilă globală folosită pentru a stoca atributele tokenilor. Această variabilă este de tipul **YYSTYPE** şi este de obicei un union de diferite câmpuri pentru tokeni diferiți.
- **yylloc** este o variabilă globală care stochează locația, linia şi coloana token-ului. Este de tipul **YYLTYPE**.

Acțiuni Lex

- Există o acțiune predefinită: dacă programul detectează un şir de caractere care nu se potriveşte cu nici un *pattern*, atunci se execută acțiunea predefinită care constă din copierea la ieşirea standard a şirului respectiv. De aceea, dacă se doreşte evitarea acestui lucru, trebuie definite acțiuni pentru orice şir de caractere posibil.
- Regula care nu efectuează nimic: dacă vreau ca un şir de caractere să fie pur şi simplu ignorat nu pun ca regulă nimic, doar **;**

  Exemplu: ignorarea spațiilor albe

```
[ \t\n]    ;
```

- Rutina **yymore( )**: are ca rezultat concatenarea următoarei lexeme de cea păstrată în **yytext**.

  Exemplu: dacă definesc tokenul şir de caractere ca fiind orice caractere puse între „”, în care pot să apară şi „, dar doar precedate de \.

```
\”[^“]*      {
                  if(yytext[yyleng-1]==’\\’) yymore();
                  else printf(„sirul de caractere =%s\n“,yytext);
             }
```

Rezolvarea ambiguităților

Limbajul Lex poate trata seturi de reguli ambigue. Anume atunci când mai multe reguli sunt aplicabile Lex alege una dintre ele astfel:

1. Cel mai lung şir care se potriveşte pe un *pattern* este ales.

2. Dacă pentru cel mai lung şir care se potriveşte pe un *pattern* există mai mult de o regulă atunci se alege prima întâlnită.

Exemplu:

```
[0-9]+                printf(„numar intreg“);
[0-9]+(\.[0-9]+)?     printf(„numar fractionar“);
```

Dacă întâlneşte 923.1 va alege a doua regulă.

Atenție:

Dacă inversez ordinea regulilor va alege şi pentru orice număr întreg varianta numărului fracționar, deoarece se potriveşte pe ambele reguli şi atunci o alege pe prima.

Exemplu 2:

```
%{
     int numchar = 0, numword = 0, numline = 0;
%}
%%
\n              {numline++; numchar+=2;}
[^ \t\n]+       {numword++; numchar+= yyleng;}
.               {numchar++;}
%%
main()
{
     yylex();
     printf("Numarul de caractere = %d\n", numchar);
     printf("Numarul de cuvinte = %d\n", numword);
     printf("Numarul de linii = %d\n", numline);
}
```

Compilări şi execuție:

```
flex ex2.l

gcc -o ex2 lex.yy.c -lfl

./ex2
```

sau

```
./ex2<in.txt
```

Tema de laborator 1:

Scrieți un program Lex (Flex) care să recunoască următoarele şiruri de caractere peste alfabetul V = {0, 1}:

- Toate cuvintele care se termină în 01
- Toate cuvintele care conțin exact un 0
- Toate cuvintele care conțin un număr par de 1 şi nici un 0
- Toate cuvintele care conțin un număr par de 1
- Toate cuvintele care conțin subşirul 01
- Toate cuvintele care nu conțin subşirul 01
- Toate cuvintele care nu conțin subşirul 011

Tema de laborator 2:

- Să se transforme toate secvențele de caractere albe într-un singur caracter alb şi secvențele de newline într-un singur newline dintr-un fişier text obținând rezultatul în alt fişier text.
- Să se determine toți identificatorii, toate numerele întregi şi toate numerele reale dintr-un fişier text.
- Să se determine şirurile de caractere dintr-un fişier text ce conține un program în limbajul C.
- Să se elimine comentariile dintr-un fişier text ce conține un program în limbajul C.

Tema de laborator 3:

Completați exemplul pentru analizorul lexical din man pages corespunzător comenzii flex.

```
/* scanner for a toy Pascal-like language */
%{
  /* need this for the call to atof() below */
  #include <math.h>
%}
DIGIT        [0-9]
ID           [a-z][a-z0-9]*
%%
{DIGIT}+     {
                printf( "An integer: %s (%d)\n", yytext,
                atoi( yytext ) );
             }

{DIGIT}+"."{DIGIT}*       {
                              printf( "A float: %s (%g)\n", yytext,
                                    atof( yytext ) );
                          }

if|then|begin|end|procedure|function        {
```

```
                                printf( "A keyword: %s\n", yytext );
                                        }

{ID}         printf( "An identifier: %s\n", yytext );

"+"|"-"|"*"|"/"    printf( "An operator: %s\n", yytext );

"{"[^}\n]*"}"      /* eat up one-line comments */

[ \t\n]+           /* eat up whitespace */

.                  printf( "Unrecognized character: %s\n", yytext );

%%
main( argc, argv )
  int argc;
  char **argv;
{
       ++argv, --argc;  /* skip over program name */
       if ( argc > 0 )
              yyin = fopen( argv[0], "r" );
       else
              yyin = stdin;
       yylex();
}
```

Bibliografie:

Marinescu D, „Tehnici de compilare”
Marinescu D, „Limbaje formale şi teoria automatelor”