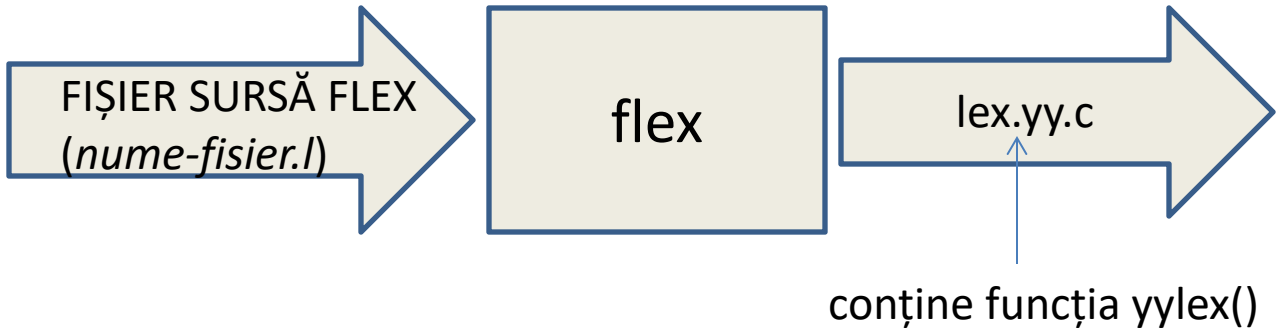
flex – partea 1

- Descriere flex (Fast LEXical Analyzer), un instrument care generează programe destinate recunoașterii diferitelor pattern-uri pentru fișiere-text (expresii regulate).
- flex poate fi considerat un generator de analizoare lexicale.
- Specificațiile ce urmează sunt extrase din *The flex Manual*, versiunea 2.5.37, 2012.
- Manualul a fost scris de Vern Paxson, Will Estes și John Millaway.


FIȘIER SURSĂ FLEX
(*nume-fisier.l*)
flex
lex.yy.c
conține funcția yylex()

Formatul fișierului de intrare pentru *flex* (*nume-fisier.l*)

Definiții

%%

Reguli

%%

Cod utilizator

Secțiunea de definiții

Secțiunea *Definiții* conține:

- Declarații de *nume* pentru expresii regulate simple, folosite pentru simplificarea specificațiilor din partea de *Reguli*
- Declararea *condițiilor de start*

Secțiunea de definiții

Declarațiile pentru pattern-uri (expresii regulate) simple au forma:

nume definiție

unde '*nume*' este un cuvânt care începe cu o literă sau underscore ('_'), urmată de zero sau mai multe litere, cifre, '_', sau '-' (dash). Definiția propriu-zisă începe de la primul caracter diferit de *whitespac*e după *nume* și continuă până la sfârșitul liniei.
Definiția va fi referită prin '{*nume*}', care va expanda '(*definiție*)'. De exemplu:

```
DIGIT    [0-9]
ID       [a-z][a-z0-9]*
```

Secțiunea de definiții

În exemplul de mai sus, DIGIT reprezintă o cifră între 0 și 9, iar ID o literă mică urmată de zero sau mai multe litere mici sau cifre.
Astfel:

{DIGIT}+"."{DIGIT}*

este echivalent cu:

([0-9])+"."([0-9])*

Secțiunea de definiții

- Un comentariu neindentat (adică o linie care începe cu ‘/*’) este copiată întocmai (*verbatim*) până la ‘*/’ în fișierul de ieșire.
- Orice text *indentat* sau orice text inclus între ‘%{’ și ‘%}’ este de asemenea întocmai copiat în ieșire (cu simbolurile %{ și %} șterse). Simbolurile %{ și %} trebuie să fie neindentate pe liniile pe care se află.
- Un bloc %top este similar unui bloc ‘%{’ ... ‘%}’, exceptând faptul că porțiunea de cod din blocul %top este relocată la *începutul* fișierului generat, înaintea oricăror definiții flex.

Secțiunea de reguli

- Secțiunea *reguli* a fișierului flex de intrare conține o serie de reguli de forma:

  pattern acțiune

  unde pattern-ul (șablonul sau expresia regulată) este neindentat iar acțiunea trebuie să înceapă pe aceeași linie.
- În secțiunea de reguli, orice text indentat sau inclus între %{ și %} ce apare înaintea primei reguli poate fi utilizat pentru a declara variabile locale pentru funcția de scanare, iar după reguli, cod care va fi executat atunci când este executată rutina de scanare (yylex()).
- În cadrul regulilor, orice text *indentat sau* text inclus între '%{' și '%}' este copiat ca atare în ieșire (cu %{ și %} șterse). Simbolurile '%{' și '%}' trebuie să apară neindentate pe liniile pe care se află.

Secțiunea de *cod utilizator*

- Această secțiune, care cuprinde cod scris de utilizator, este copiată ca atare în fișierul generat de flex, yy.lex.c
- Prezența acestei secțiuni este opțională. Dacă este omisă, atunci cel de-al doilea ‘%%’ poate fi de asemenea omis.

Comentarii în flex

- Flex suportă comentarii în stil C, adică orice text cuprins între ‘/*’ și ‘*/’ este considerat comentariu. Comentariile sunt copiate ca atare în codul generat de flex. Comentariile pot să apară oriunde, cu următoarele excepții:
- Comentariile nu pot apare în secțiunea *Reguli* atunci când flex așteaptă o expresie regulată. Aceasta înseamnă că un comentariu nu poate să înceapă la începutul unei linii sau imediat după o listă de stări ale scanner-ului.
- Comentariile nu pot apare într-o linie ‘%option’ din secțiunea de *Definiții*.
- O regulă simplă pentru a scrie un comentariu: începeți un comentariu pe o linie nouă, cu unul sau mai multe spații înainte de ‘/*’. Această regulă funcționează oriunde în fișierul de intrare flex.

Pattern-uri (șabloane) în flex

- Pattern-urile în fișierul de intrare flex sunt scrise cu ajutorul unui set extins de expresii regulate. Acestea sunt:
- x – folosit pentru caracterul 'x'
- . – folosit pentru orice caracter (byte), exceptând newline
- [xyz] – folosit pentru o *clasă de caractere*; în acest caz se referă la un caracter care poate fi x, y sau z.

Pattern-uri (șabloane) în flex

- [abj-oZ] – o “clasă de caractere" cu interval; se potrivesc caracterele a, b, orice caracter între j și o inclusiv, și Z
- [^A-Z] – o “clasă de caractere negate", adică orice caracter cu excepția celor din această clasă. În acest caz, orice caracter EXCEPTÂND o literă mare.
- [^A-Z\n] – orice caracter EXCEPTÂND o literă mare sau newline
- [a-z]{-}[aeiou] – toate consoanele (caractere mici, fără vocale).

Pattern-uri (șabloane) în flex

- r* – zero sau mai multe apariții ale lui r, unde r este orice expresie regulată
- r+ – una sau mai multe apariții ale lui r
- r? – zero sau o apariție a lui r (aceasta înseamnă “r opțional”)
- r{2,5} – între 2 și 5 apariții ale lui r
- r{2,} – două sau mai multe apariții ale lui r
- r{4} – exact 4 apariții ale lui r

Pattern-uri (șabloane) în flex

- {*name*} – expandarea definiției '*name*' (vezi secțiunea *Definiții*).
- "[*xyz*] \"*foo*" – șirul literal: [*xyz*]"foo
- \X – dacă 'X' este 'a', 'b', 'f', 'n', 'r', 't' sau 'v', atunci interpretarea ANSI-C pentru '\x'. Altfel, literalul 'X' (utilizat pentru a scrie ca atare caractere precum '"' în interiorul unui șir sau operatorul '*')
- \0 – caracterul NUL (cod ASCII 0)
- \123 – caracterul cu valoarea 123 în octal
- \x2a – caracterul cu valoarea 2a în hexazecimal
- (r) – aceeași ca și 'r'; parentezele sunt utilizate pentru a schimba precedența

Pattern-uri (șabloane) în flex

- (?*r-s:pattern*) – aplică opțiunea ‘r’ și omite opțiunea ‘s’ când interpretezi *pattern*. Opțiunile pot fi zero sau mai multe caractere ‘i’, ‘s’ sau ‘x’.
- i înseamnă case-insensitive.
- -i înseamnă case-sensitive.
- s alterează înțelesul lui ‘.’ de a se potrivi cu orice byte.
- -s alterează înțelesul lui ‘.’ de a se potrivi cu orice byte cu exceptia ‘\n’.
- x semnifică ignorarea de comentarii și whitespace în pattern-uri. Whitespace este ignorat până ce nu apare ca backslash-escaped, conținut între ghilimele (""), sau până ce nu apare într-o clasă de caractere.
- Următoarele pattern-uri sunt valide:

Pattern-uri (șabloane) în flex

```
(?:foo)           la fel ca  (foo)
(?i:ab7)          la fel ca  ([aA][bB]7)
(?-i:ab)          la fel ca  (ab)
(?s:.)            la fel ca  [\x00-\xFF]
(?-s:.)           la fel ca  [^\n]
(?ix-s: a . b)    la fel ca  ([Aa][^\n][bB])
(?x:a  b)         la fel ca  ("ab")
(?x:a\ b)         la fel ca  ("a b")
(?x:a" "b)        la fel ca  ("a b")
(?x:a[ ]b)        la fel ca  ("a b")
(?x:a
    /* comment */
    b
    c)            la fel ca  (abc)
```

Pattern-uri (șabloane) în flex

- (?# comment) – comentariu; omite orice cuprins între ‘()’. Primul caracter ‘)’ întâlnit finalizează pattern-ul. Nu este posibil ca acest comentariu să conțină caracterul ‘)’. Acest tip de comentariu poată să cuprindă mai multe linii.
- rs – expresia regulată ‘r’ urmată de expresia regulată ‘s’; este *concatenarea* celor două expresii
- r|s –‘r’ sau ‘s’ (reuniunea)
- r/s – ‘r’ doar dacă este urmată de ‘s’
- ^r – ‘r’, însă doar la început de linie
- r$ – ‘r’, însă doar la sfârșit de linie (adică chiar înainte de newline). Echivalent cu ‘r/\n’.

Pattern-uri (șabloane) în flex

- <s>r – ‘r’, dar doar în condiția de start s
- <s1,s2,s3>r – ‘r’ într-una din condițiile de start s1, s2 sau s3.
- <*>r – ‘r’, în orice condiție de start, chiar una exclusivă.
- <<EOF>> – end-of-file.
- <s1,s2><<EOF>> –end-of-file într-una din condițiile de start s1 sau s2

Pattern-uri (șabloane) în flex – observații

- În interiorul unei clase de caractere, toți operatorii își pierd înțelesul, cu excepția lui escape (‘\’) și a operatorilor de clasă: ‘-’, ‘]’,’[’ și a operatorului ‘^’ la începutul clasei.
- Expresiile regulate listate într-un fișier de intrare sunt grupate în acord cu precedența, de sus în jos, de la precedența cea mai mare către cea mai mică. Cele grupate împreună au aceeași precedență. De exemplu:

  $foo|bar*$ este la fel cu $(foo)|(ba(r*))$
- Un șir care se potrivește cu acest pattern este *fie* șirul ‘foo’ *fie* șirul ‘*ba*’ urmat de zero sau mai mulți de ‘r’.

Pattern-uri (șabloane) în flex – observații

- În plus față de caractere și domenii de caractere, clasele de caractere pot conține *expresii de clase de caractere*. Acestea sunt expresii cuprinse între ‘[:’ și ‘:]’ (care trebuie să apară între ‘[’ și ‘]’, alături, eventual, de alte elemente). Exemple de expresii de clase de caractere:
- [:alnum:] [:alpha:] [:blank:]
- [:cntrl:] [:digit:] [:graph:]
- [:lower:] [:print:] [:punct:]
- [:space:] [:upper:] [:xdigit:]

- Următoarele clase de caractere sunt echivalente:

- [[:alnum:]]
- [[:alpha:][:digit:]]
- [[:alpha:][0-9]]
- [a-zA-Z0-9]

Pattern-uri (șabloane) în flex – observații

- Flex permite negarea expresiilor de clase de caractere prin utilizarea lui ‘^’ , ca în:
- [:^alnum:] [:^alpha:] [:^blank:]
- [:^cntrl:] [:^digit:] [:^graph:]
- [:^lower:] [:^print:] [:^punct:]
- [:^space:] [:^upper:] [:^xdigit:]
- Operatorul ‘{-}’ este folosit pentru diferența dintre două clase de caractere. De exemplu, ‘[a-c]{-}[b-z]’ reprezintă toate caracterele din clasa ‘[a-c]’ care nu sunt în clasa ‘[b-z]’ (ceea ce înseamnă, în cazul de față, doar caracterul ‘a’). Operatorul ‘{-}’ este asociativ la stânga, deci ‘[abc]{-}[b]{-}[c]’ este același cu clasa ‘[a]’. Atenție: să nu creați mulțimea vidă, care nu se potrivește niciunui șir.
- Operatorul ‘{+}’ calculează reuniunea a două clase de caractere. De exemplu, ‘[a-z]{+}[0-9]’ este la fel cu ‘[a-z0-9]’. Acest operator este util atunci când este precedat de rezultatul unei diferențe, ca în: ‘[[:alpha:]]{-}[[:lower:]]{+}[q]’, care este echivalent cu ‘[A-Zq]’.

Cum se potrivesc șirurile de intrare cu pattern-urile

Atunci când este executată funcția scanner (yylex(), obținută cu flex), aceasta analizează intrarea căutând șirurile care se potrivesc cu pattern-urile (expresiile regulate declarate în fișierul flex). Dacă găsește mai multe potriviri, o alege pe aceea care se potrivește cu textul cel mai lung (amintiți-vă descrierile lexicale orientate dreapta). Dacă pentru un același șir de intrare sunt găsite potriviri cu mai multe pattern-uri, este acceptată prima regulă listată în fișierul flex, de sus în jos (cu precedența cea mai mare).

Odată ce a fost determinetă potrivirea, textul corespondent (numit *token*) este făcut accesibil prin pointerul caracter, global **yytext**, iar lungimea sa în variabila globală de tip integer **yyleng**. *Acțiunea* corespondentă pattern-ului este apoi executată, după care restul input-ului este scanat pentru a găsi următoarea potrivire (*token*).

Dacă nu este găsită nicio potrivire, atunci este executată **regula implicită (*default rule*)** : caracterul următor din input este considerat ca potrivindu-se și copiat în output-ul standard. Astfel, cel mai simplu input valid pentru flex este:

```
%%
```

care generează un scanner care doar copiază input-ul (caracter cu caracter) în ieșire.

Cum se potrivesc șirurile de intrare cu pattern-urile

- yytext poate fi definit în două moduri: fie *pointer* caracter sau *array* de caractere. Pentru aceasta se folosesc directivele flex %pointer sau %array în secțiunea de *Definiții* flex. Prin absență, este considerat %pointer, dacă nu este utilizată opțiunea ‘-l’ pentru combatibilitatea cu lex, caz în care yytext este array. Avantajul utilizării %pointer este rapiditatea scanării.
- Avantajul lui %array este că poate fi modificat yytext și apoi poate fi apelată unput() pentru a nu distruge yytext.
- Declarația %array definește yytext ca un array de YYLMAX caractere, care prin absență ia o valoare suficient de mare.
- Nu poate fi utilizat %array cu clase C++

Acțiunile

- Fiecare pattern dintr-o regulă (din Secțiunea *Reguli*) are o *acțiune* corespondentă, care poate fi orice instrucțiune C. Pattern-ul se sfârșește la primul caracter whitespace non-escaped; restul liniei reprezintă acțiunea corespondentă. Dacă acțiunea este vidă, atunci șirul (token-ul) care se potrivește cu pattern-ul corespondent (de pe aceeași linie) este pur și simplu descărcat. De exemplu, în următorul exemplu toate aparițiile șirului 'zap me' sunt șterse:

```
%%
"zap me"
```

- În exemplul de mai sus toate celelalte caractere din input sunt copiate în output, deoarece se potrivesc cu regula implicită (*default rule*).
- Mai jos este un program care comprimă mai multe blank-uri și tab-uri într-un singur blank:

```
%%
[ \t]+        putchar( ' ' );
[ \t]+$       /* ignore this token */
```

Acțiunile

- Dacă acțiunea începe cu ‘{’, atunci acțiunea se întinde până este întâlnit ‘}’, deci acțiunea poate fi scrisă pe mai multe linii. Flex recunoaște șirurile și comentariile C, dar permite ca acțiunile să înceapă cu ‘%{’ care se va încheia la următoarele ‘%}’ (atunci când folosim în interiorul acțiunii ‘{‘, ‘}’).
- Acțiunea care constă doar din bara verticală (‘|’) semnifică “**aceeași cu regula pentru următoarea regulă”.**
- Acțiunile pot include cod oarecare C, incluzând instrucțiuni return pentru a returna o valoare pentru yylex(). De câte ori yylex() este apelată, ea continuă procesarea token-ilor din punctul în care a rămas până când fie întâlnește *end_of_file*, fie *return*.
- Acțiunile pot modifica yytext (prin adăugarea de caractere la sfârșit), exceptând lungimea lui (aceasta ar produce suprascrierea caracterelor din input). Aceasta nu se aplică când folosim %array, caz în care yytext poate fi modificat în orice mod.
- Acțiunile pot modifica yyleng, exceptând cazul când se utilizează yymore()

Directive ce pot fi incluse în Acțiuni

- Există un număr de directive speciale ce pot fi incluse în acțiuni:

ECHO – copiază yytext în output-ul scanner-ului.

BEGIN – când este urmată de numele unei condiții de start, plasează scanner-ul în starea corespondentă condiției de start.

REJECT – direcționează scanner-ul de a trece la "următoarea cea mai bună" regulă care se potrivește input-ului (sau unui prefix din input) actual. Regula este aleasă așa cum a fost deja descris, iar yytext și yyleng sunt setate adecvat. În exemplul următor, simultan sunt contorizate cuvintele din input și este apelată funcția special() ori de câte ori cuvântul 'frob' este întâlnit:

```
            int word_count = 0;
%%
frob        special(); REJECT;
[^ \t\n]+   ++word_count;
```

- Fără REJECT, frob nu ar fi contorizat ca și cuvânt

Directive ce pot fi incluse în Acțiuni

- Pot fi utilizate mai multe directive REJECT, fiecare alegând următoarea cea mai bună alegere a regulii curent active. De exemplu, când scanner-ul scanează token-ul ‘abcd’, el va scrie ‘abcdabcaba’ în output:

```
%%
a       |
ab      |
abc     |
abcd  ECHO; REJECT;
.|\n    /* ”mănâncă” orice alt caracter */
```

- Primele trei reguli au aceeași acțiune asociată ca cea de-a patra, deoarece este utilizat simbolul ‘|’.

Funcții ce pot fi incluse în Acțiuni

- yymore() – spune scanner-ului ca la următoarea potrivire token-ul corespondent să fie *adăugat* la yytext în loc să îl înlocuiască. De exemplu, dat fiind input-ul ‘mega-kludge’, următorul scanner va produce în output ‘mega-mega-kludge’ :

```
%%
mega-    ECHO; yymore();
kludge   ECHO;
```

- Primul ‘mega-’ se potrivește intrării și este scris în output. Apoi ‘kludge’ se potrivește, dar anteriorul ‘mega-’ apare încă la începutul lui yytext, deci ECHO pentru ‘kludge’ va avea ca efect scrierea (adăugarea) lui ‘mega-kludge’ în ieșire.

Funcții ce pot fi incluse în Acțiuni

- yyless(n) returnează tot, mai puțin primele n caractere ale token-ului curent, în stream-ul de intrare, de unde vor fi re-scanate atunci când scanner-ul caută următorul token. yytext și yyleng sunt ajustate adecvat (yyleng va fi egal cu n). De exemplu, pentru input-ul ‘foobar’, cu următoarele instrucțiuni se va scrie în ieșire ‘foobarbar’:

```
%%
foobar    ECHO; yyless(3);
[a-z]+    ECHO;
```

- Argumentul 0 pentru yyless() va avea ca efect ca întregul șir să fie scanat din nou. Până când nu veți modifica modul cum scanner-ul va procesa input-ul (utilizând BEGIN, de exemplu), aceasta va genera intrarea într-un loop infinit. Observați că yyless() este macro și poate fi utilizat doar de fișierul de intrare pentru flex.

Funcții ce pot fi incluse în Acțiuni

- unput(c) pune înapoi caracterul c în input stream. Va fi următorul caracter scanat.
- input() citește următorul caracter din input stream. Dacă scanner-ul este compilat cu C++, atunci input() este referită ca **yyinput()**.
- YY_FLUSH_BUFFER; descarcă buffer-ul intern al scanner-ului, astfel ca data următoare când scanner-ul asteaptă să potrivească un token, mai întâi va reumple buffer-ul prin apelarea YY_INPUT()
- yyterminate() poate fi utilizată în loc de instrucțiunea *return* într-o acțiune. Finalizează scanarea și întoarce 0 programului apelant. yyterminate() este apelată implicit (*by default*) atunci când este întâlnit *end-of-file*. Este un macro și poate fi redefinit.

SCANNER-UL GENERAT

- Ieșirea produsă de flex este fișierul lex.yy.c, care conține: funcția de scanare yylex(), un anumit număr de tabele utilizate la potrivirea token-ilor (de fapt tabelele corespunzătoare AFD ce recunosc limbajele descrise de expresiile regulate cuprinse în fișierul de intrare pentru flex), și anumite rutine auxiliare și macrouri. Implicit, yylex() este declarată:

```
int yylex()
    {
        ... diferite definiții și acțiuni ...
    }
```

- Dacă mediul folosit suportă funcții prototip, atunci va fi int yylex(void). Această declarație poate fi modificată cu ajutorul macroului YY_DECL

EXEMPLUL 1

Următorul input pentru *flex* este specificația unui scanner care, când întâlnește șirul ‘username’ îl va înlocui cu numele de login al utilizatorului:

```
%%
username    printf( "%s", getlogin() );
```

Implicit, orice text care nu se potrivește scanner-ului flex este copiat în output, astfel că efectul scanner-ului de mai sus este de a copia fișierul de intrare în output, cu fiecare apariție a lui ‘username’ expandată. În acest input, există doar o regulă. ‘username’ este pattern-ul iar ‘printf’ este acșiunea. Simbolul ‘%%’ marchează începutul regulii asociate.

EXEMPLUL 2

```
        int num_lines = 0, num_chars = 0;
%%
\n      ++num_lines; ++num_chars;
.       ++num_chars;
%%
int main()
        {
        yylex();
        printf( "# of lines = %d, # of chars = %d\n",
                num_lines, num_chars );
        }
```

Acest scanner contorizează numărul de caractere și numărul de linii din input. Nu produce la ieșire decât raportul final al numărului de caractere și linii. În prima linie sunt declarate variabilele globale num_lines și num_chars, care sunt accesibile atât din yylex() cât și din main(), declarată după al doilea '%%'. Sunt 2 reguli, una pentru potrivirea cu newline ('\n'), când se incrementează atât contorul de linie, cât și cel de caractere, și o regulă pentru potrivirea cu orice caracter mai puțin newline (indicată de '.').

EXEMPLUL 3

```
/* scanner pentru un mini-limbaj Pascal-like */

  %{
  /* este nevoie pentru apelul lui atof() de mai jos*/
  #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 );
```

```
%%

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

        yylex();
        }
```