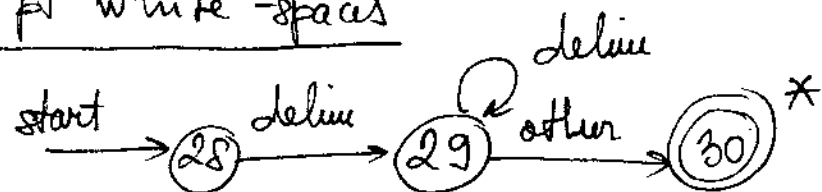


D.T. pt white-spaces



Implementarea unei D.T. (diag. tranziții)

Fiec. stare \rightarrow asoc. un segment de cod.

De \exists arce care pleacă din starea rep. \rightarrow citit unu. caracter și selectată calea.

nextchar() \rightarrow

- 1) citește un caracter din buff.
- 2) avansează fwd. pointerul
- 3) întoarce caract. citit

Pt. a întoarce un at. lexical \rightarrow var. globală \rightarrow lex_value

clasa at. lexical este întoarsă de funcția principală a an. lexical \rightarrow main()

```
int store = 0, start = 0;
```

```
int lex_val;
```

```
int fail() {
```

```
    fwd_ptr = leg_ptr;
```

```
    switch (start) {
```

```
        case 0: start = 9; break;
```

```
        case 9: start = 12; break;
```

```
        case 12: start = 20; break;
```

```
        case 20: start = 25; break;
```

```
        case 25: recover(); break;
```

```
        default: /* err. comp. */
```

```
    }  
    return start;
```

```
}
```

```
Atom runAtom() {
```

```
    while (1) {
```

```
        switch (state) {
```

```
            case 0: c = nextchar();
```

```
                if (c == '\back' || c == '\tab' || c == '\n') {
```

```
                    state = 0;
```

```
                    beg_ptr ++;
```

```
                }  
                else if (c == '<') state = 1;
```

```
                else if (c == '=') state = 5;
```

```
                else if (c == '>') state = 6;
```

```
                else state = fail();
```

```
        break;
```

/* case while 1-8 */

case 9: c = nextchar();

if (isletter(c)) state = 10;

else state = fail();

break;

case 10: c = nextchar();

if (isletter(c) || isdigit(c)) state = 10;

else state = 11;

break;

case 11: retract(1); instal_id();

return getAt();

/* case while 12-24 */

case 25: c = nextchar();

if (isdigit(c)) state = 26;

else state = fail();

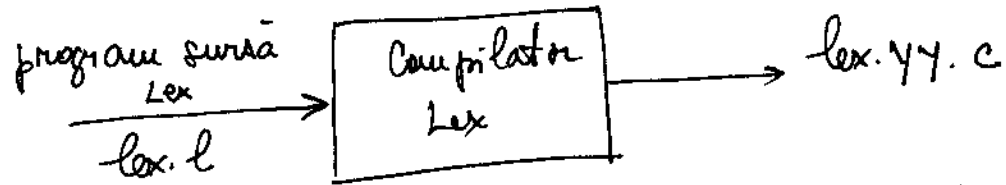
break;

```
case 26: c = nextchar();  
    if (isdigit(c)) state = 26;  
    else state = 27;  
    break;
```

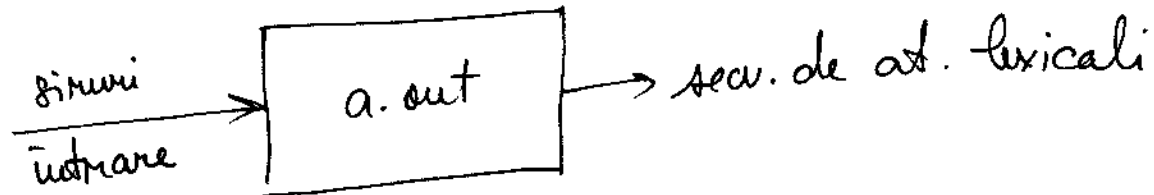
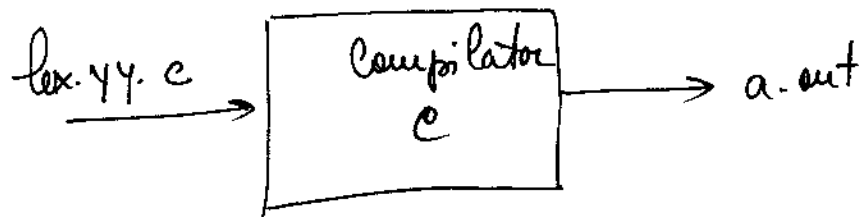
```
case 27: retract(1); instal-num();  
    return NUM;
```

```
    }  
    }  
    }
```

Limbaj pentru specificarea au. lexicale LEX



o reprezentare tabelară
a diag. de tranziții
construite din expresiile reg.
din lex.l + fcti standard



Specificare lex.

Un program lex \rightarrow 3 părți:

declarații

% %

reguli de traducere

% %

funcții auxiliare

Declarații \rightarrow def. variabile, constante, def. regulate

Reguli de traducere:

p1 3 acțiune 1

p2 3 acțiune 2

:

pn 3 acțiune n

unde $p_i \rightarrow$ exp. regulată

acțiunii \rightarrow fragment de cod care descrie ce acțiune trebuie să
execute au. lexical când este găsită o potrivire cu p_i

Fcti auxiliare \rightarrow fcti apelate în acțiuni

Obs:

An. lexical întoarce doar at. lexical. Pt a întoarce o valoare atribut întoarce
au info despre lexemă \Rightarrow var. globală yybval.

Spec. lex.

%} /* def. constante

LT, LE, EQ, NE, GT, GE, IF, THEN, ELSE, ID, NUM, OPREL */

%}

/* def. regulate */

delim [+ \ m]

ws { delim } +

lit [A - Z a - z]

cif [0 - 9]

id { lit } ({ lit } | { cif }) *

num { cif } + (\ . { cif } +) ? (E [+ \ -] ? { cif } +) ?

% %

{ ws } { /* inicio archivo */ }

if { return IF; }

then { return THEN; }

else { return ELSE; }

{ id } { yy.bval = instal - id(); return ID; }

{ num } { yy.bval = instal - num(); return NUM; }

"<" { yy.bval = LT; return OPREL; }

"<=" 3 yy lval = LE ; return OPREL; }

i
%%

instal_id() 3 /* folie care pune lexema (sir de caract.) al cărei primul caracter
este indicat de yytext în a cărei lungime este yy-leng, în tab. de
simboluri și întoarce intr. din tabelă */

instal_num() 3 /* similar pt. a adauga o const. numerică în T-S. */

Obs:

- Tot ceea ce este între %? și %? este copiat direct în lex.yy.c și nu este tratat
ca parte a def. regulate din regulile de translație.
- ac. lucru ⇒ folii proc. def. în a 3-a secțiune
- \. ⇒ pot, altfel reprez. clasa tuturor caracterelor except newline
- \- ⇒ -, altfel simbol de interval [A-Z]

yybval \rightarrow def. in lex.yy.c

yytext \rightarrow pointer către primul caracter din lexemă (leg-ptr)

yylen \rightarrow lungimea lexemei (am folosit fwd-ptr).

Limbaje Independente de Context

Gramatici Indep. de Context

L. regulate $\left\{ \begin{array}{l} \text{acceptori} \rightarrow A.F. \\ \text{generatori} \rightarrow E.R. \end{array} \right.$

$a(a^*uf^*)b$

Dc. $S \rightarrow$ nou simbol interpretat ca sir in limbaj
 $M \rightarrow$ simbol cresp. (a^*uf^*)

$S \rightarrow aMb \Rightarrow$ regulă

$M \rightarrow A$

$M \rightarrow B$

$A \rightarrow aA \mid \epsilon$

$B \rightarrow bB \mid \epsilon$

Pornesc cu S , găsesc un simbol care în sirul generat de S , care apare la stg. într-una dintre reguli. Inloc. simbol cu simbol care apare în dr. regulii, etc.

$S \rightarrow aAb \rightarrow aAb \rightarrow aaAb \rightarrow aaaAb \rightarrow aaab$

Def. autoisomorfism \rightarrow gramatică (G)

? G. indep. de context

$(aA)Ab$

✓ contextul lui A

$A \rightarrow aA \Rightarrow A$ se poate înlocui prin simbolul aA indiferent de contextul lui A.

Def.

o gramatică indep. de context, G, este un tuple (V, Σ, R, S) , unde:

$V \rightarrow$ alfabet

$\Sigma \subseteq V \rightarrow$ mulțimea simb. terminale

$R \subseteq (V - \Sigma) \times V^* \rightarrow$ mulțimea regulilor

\uparrow submulțime finită

$S \in V - \Sigma \rightarrow$ simbolul de start

Obs:

$V - \Sigma \rightarrow$ mult. simb. neterminale (neterminali)