Vad gör en lexikalanalys?

Ex indata:

```
apa := 1 + 2 * bepa
```

- **ANALYS:**
  - **Lexikalanalys**
    Känner igen "orden":
    ```
    [L_id "apa", L_assign,
     L_const 1, L_plus, L_const 2, L_mult, L_id "bepa" ]
    ```
  - **Syntaxanalys**
    ```
           E_assign
           /      \
    E_id "apa"   E_plus
                 /    \
          E_const 1   E_mult
                      /    \
               E_const 1  E_id "bepa"
    ```
  - **Semantisk analys** Kolla att `apa` och `bepa` är definierade innan, med rätt typ, exvis `int`.

Vad gör en lexikalanalys?

Lexikalanalysen är alltså första hälften av text-analysen, *parsningen*, av indata.

Lexikalanalysatorns huvuduppgift:

att gruppera tecken till större enheter, **lexikala symboler** (**terminala symboler**, **tokens**, **"lexical tokens"**)

Andra uppgifter (kan vara):

- eliminera onödig information som tex blanka och kommentarer,
- stoppa in identifierare i symboltabell eller strängtabell (i en imperativ kompilator).
- Hålla reda på radnr och filnamn för felutskrifter
- ...

Imperativt: lexikalanalysatorn är ofta en subrutin till parsern.

Varför dela upp parsningen i två nivåer?

dvs **lexikalanalysen** och "den riktiga" **syntaxanalysen** ?

- Det underlättar för parsern att få arbeta på en högre nivå.
  Ex:
  ```
  ifrågasättandegrad := 2 ;
  ^
  ```
  Är detta början på en if-sats?
  Ex:
  ```
  /* en kommentar */
  ^
  ```
  Är detta division?
- Något annorlunda "igenkänningsproblem": tex reserverade ord är ofta en delmängd av identifierare.
- Bättre modularitet
- Kan bli effektivare att separera text-arbetet från resten av kompilatorn
- .... (finns säkert fler bra skäl!)

Formalism

En typisk lexikalanalys är ganska enkel att implementera "för hand".

Men vi kommer att studera lexikalanalys i termer av kraftfull formalism, **reguljära uttryck** och **ändliga automater**

- liknande formalism är användbar i det vidare parsningsproblemet
- grunden för verktyg för automatisk konstruktion av lexikalanalysatorer (lex, alex, flex, etc.)

Vanliga typer av lexikala symboler

Definition: ett **lexem** ar den teckensträng som ger en viss lexikal symbol.

symboltyp	ex lexem	ex lexikal symbol	ev associerat värde
identifierare	`apa`	ID	"apa"
	`bepa`	ID	"bepa"
reserverade ord	`if`	IF	
	`then`	THEN	
tal	`14`	INTCONST	14
	`3.14`	FLOATCONST	3.14
	`.4e-12`	FLOATCONST	0.4e-12
symboler	,	COMMA	
	(	LPAR	
	/=	NOTEQ	

Vad är *inte* lexikala symboler?? (typiskt)

kommentarer
```
/* Här går det undan!  */
// Här går det undan!
```

Preprocessor-direktiv:
```
#include "defs.h"
#define FEL printf("hupp!")
 FEL
```

"osynliga" tecken: blanktecken, tabulatortecken, nyrader

Hur beskriva lexikala symboler?

- **Naturligt språk:** Ex identifierare i C eller Java:
  "An identifier is a sequence of letters and digits; the first character must be a letter. The underscore _ counts as a letter. Upper- and lowercase letters are different."
- **Reguljära uttryck/definitioner:**
  Ex indata till flex:

```
letter =  [A-Za-z_]
digit  =  [0-9]
ident  =  {letter}({letter}|{digit}|_)*
```


Reguljära uttryck: symbol som står för sig själv

Ex:

a (där tecknet **a** tillhör alfabetet)

$L(\mathbf{a}) \equiv \{\text{"a"}\}$

$L(M)$ = det språk, dvs den mängd strängar, som det reguljära uttrycket M beskriver.

Reguljära uttryck: alternativ $M|N$

(där M och N är reguljära uttryck)

$L(M|N) \equiv L(M) \cup L(N)$

Ex:

a|**b**|**c**

beskriver språket

$\{\text{"a"}, \text{"b"}, \text{"c"}\}$

dvs.

$L(\mathbf{a}|\mathbf{b}|\mathbf{c}) \equiv \{\text{"a"}, \text{"b"}, \text{"c"}\}$

Reguljära uttryck: sammansättning (konkatenering) $M \cdot N$

$L(M \cdot N) \equiv \{s_m s_n \mid \forall s_m \in L(M), s_n \in L(N)\}$

kartesisk produkt, dvs “alla kombinationer”

Ex:

$(\mathbf{a}|\mathbf{b}) \cdot (\mathbf{c}|\mathbf{d})$

beskriver språket

$\{\texttt{"ac"}, \texttt{"bc"}, \texttt{"ad"}, \texttt{"bd"}\}$

Ofta utelämnar vi $\cdot$, tex.

$L(\mathbf{ab}|\mathbf{c}) \equiv \{\texttt{"ab"}, \texttt{"c"}\}$

Reguljära uttryck: tomma strängen ϵ

beskriver språket $\{\texttt{""}\}$

Ex:

$(\mathbf{a} \cdot \mathbf{b})|\epsilon$

$\{\texttt{""}, \texttt{"ab"}\}$

Reguljära uttryck: repetition $M^\star$

$M^\star \equiv \epsilon \mid M \mid M \cdot M \mid M \cdot M \cdot M \mid \ldots$

eller, om man ska vara riktig stringent! :

$M^\star \equiv \epsilon \mid M \cdot M^\star$

dvs 0 eller flera följder av någon sträng i $L(M)$

(ger alltid en oändligt mängd!)

$M^\star$ kallas ibland Kleene-höljet av M.

Ex:

$(\mathbf{a} \cdot (\mathbf{a}|\mathbf{b}))^\star$

beskriver språket

$\{\texttt{""}, \texttt{"aa"}, \texttt{"ab"}, \texttt{"aaaa"}, \texttt{"aaab"}, \texttt{"abaa"}, \texttt{"abab"}, \ldots\}$

Reguljära uttryck: sammanfattning, plus några förkortade skrivsätt

a	Ett ordinärt tecken som står för “sig själv”.
ϵ	Den tomma strängen.
	Dito!
$M\|N$	Alternativ.
$M \cdot N$	sammansättning (konkatenering)
MN	Dito!
$M^\star$	repetering, 0 eller flera gånger
M^+	repetering, 1 eller flera gånger, dvs $M \cdot M^\star$
$M?$	valfri, 0 eller 1 förekomst, dvs $\epsilon\|M$
[**axz**]	(**a**|**x**|**z**)
[**a** − **d**]	(**a**|**b**|**c**|**d**)
[**a** − **zA** − **Z**]	går bra att räkna upp flera
.	vad som helst utom nyrad (lex/alex/flex konvention)
`"a.+*"`	En sträng inom citationstecken står för strängen själv

Reguljära uttryck för att specificera lexikala symboler

Exempel indata till lex/flex/ML-yacc/...

```
if                 { returnera symbol IF }
[a-z][a-z0-9]*     { returnera symbol ID }
[0-9]+             { returnera symbol NUM }
("//".*"\n")       { kommentar: fortsätt med nästa }
"\n"|"\t"|" "      { vitt tecken: fortsätt med nästa }
.                  { signalera fel }
```


Flertydigheter

```
if                 { returnera symbol IF }
[a-z][a-z0-9]*     { returnera symbol ID }
[0-9]+             { returnera symbol NUM }
("//".*"\n")       { kommentar: fortsätt med nästa }
"\n"|"\t"|" "      { vitt tecken: fortsätt med nästa }
.                  { signallera fel }
```

Dessa regler är *flertydiga*:

Vad betyder "`if8` " ?

- IF ("if"), NUM ("8"), eller
- ID ("if8"), eller
- ID ("if"), NUM ("8"), eller ... ?

Två regler för att "entydliga" (disambiguate)

Längsta matchning: Den längsta strängen som matchar *någon* regel tas som lexem.

Ex: "`if8`" blir identifierare, eftersom det är de enda regel som matchar detta lexem.

Regel-prioritet: Om det finns *flera* längsta matchar, tas den regel som kommer först i listan.

Ex: "`if`" matchar både IF och ID, men IF tas, eftersom den kommer först.

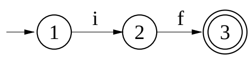
Finita (ändliga) automater

Finita automater kan känna igen strängar i språk definierade av reguljära uttryck.

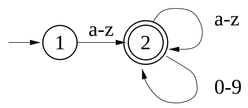
En ändlig automat har:

- En ändlig mängd **tillstånd**,
- **bågar** mellan par av tillstånd, märkta (labeled) med tecken i in-alfabetet,
- ett av tillstånden är **starttillstånd**,
- ett eller flera av tillstånden är **sluttillstånd**.

(Ofta numreras tillstånden för att bekvämt kunna referera till dem.)



känner igen IF



känner igen ID

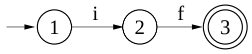
`a-z` betyder egentligen 26 bågar!

Finita automater känner igen reguljära uttryck

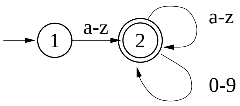
En DFA (deterministisk finit automat) **accepterar** en sträng på följande sätt:

- starta i starttillståndet
- för varje tecken i in-strängen, följ den båge som är märkt med det tecknet.
- Efter att ha gjort n övergångar för en sträng av längd n, om automaten då är i ett sluttillstånd, så är strängen igenkänd av den DFA:n.

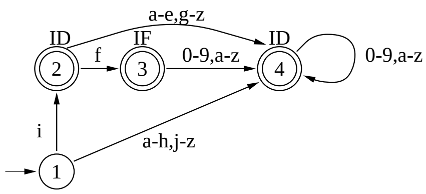
Kombinerade automater



känner igen IF



känner igen ID



känner igen ID eller IF

Att exekvera en finit automat

- **uppenbart:**
  koda automaten som en två-dimensionell array,
  $Tillstånd \times Char \rightarrow Tillstånd$,
  samt en endimensionell array
  $Tillstånd \rightarrow (Lexsym \cup \texttt{NoLexsym})$.
- **inte lika uppenbart:** P.g.a. längsta-match-regeln: Bara för att automaten befinner sig i ett slut-tillstånd betyder inte det nödvändigtvis att just den lexikala symbolen skall returneras, utan p.g.a. längsta-match-regeln ska man forsätta att försöka hitta en matching, dvs exekvera automaten.

  Misslyckas det, skall maskineriet returnera det tillstånd/lexem från senast passerat sluttillstånd, som man alltså måste minnas.

  Lämnas till den intresserade! sid 20 i boken.

Automatisk konstruktion av lexikalanalysatorer (alex, flex, etc)

NFA (nondeterministic finite automaton)

1. För varje reguljärt uttryck, konstruera en NFA (lätt)
2. Kombinera dessa till en stor NFA (lätt)
3. Konvertera denna till en DFA (svårt!)
4. Konstruera tabeller för automaten, samt kod för aktioner (lätt)

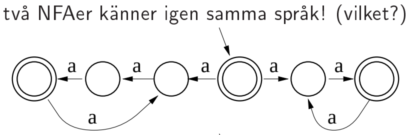
Icke-deterministiska finita automater

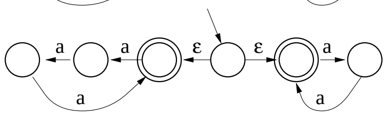
NFA (nondeterministic finite automaton)

- Från visst tillstånd, får ha samma tecken på flera ut-bågar,
- får ha ϵ på båge, som då exekveras utan att "konsumera" något tecken.

En NFA "måste alltid gissa rätt" vilken väg den ska ta för att acceptera viss sträng.

Följande två NFAer känner igen samma språk! (vilket?)





Alex

Alex är en lexikalanalysgenerator för Haskell. Den påminner om lex, en generator för C från UNIX barndom.

Indata till alex är en fil med reguljäruttryck och vilka token dessa ska generera och utdata är en Haskell-modul som gör det.

Indata till alex:

```
%{
-- Haskell-kod
%}
definitioner

"namn"/"namn":-

<namn>    := regexp
...

%{
-- Mer Haskell-kod
%}
```


Alex, exempel

```
%{
module Token where
%}
{ ^d = 0-9 }       -- digits
{ ^l = [a-zA-Z] }  -- alphabetic letters
"tokens_lx"/"tokens_acts":-

<>        ::= ^w+                -- whitespace
<>        ::= ^/^/.*             -- comments
<semi>    ::= ^;           %{ semi  p s = Semi p %}
<if'>     ::= if           %{ if'   p s = If p %}
<else'>   ::= else         %{ else' p s = Else p %}
<name>    ::= ^l[^l^d^_^]* %{ name  p s = Name p s %}
<litint>  ::= ^d+

%{
litint       p s = LitInt p (read s)
data Token
        = Semi Posn
        | If Posn
        | Else Posn
        | Name Posn String
        | LitInt Posn Int
        | Err       Posn
        deriving (Eq,Show)
```



```
tokens :: String -> [Token]
tokens inp = scan tokens_scan inp

tokens_scan = load_scan (tokens_acts,stop_act) tokens_lx
        where
           stop_act p ""  = []
           stop_act p inp = [Err p]

%}
```


(exempel som dras för hand på tavlan:)

Två reguljära uttryck:

```
"if"              { IF }
[a-z][a-z0-9]*    { ID }
```

(visa konstruktion av NFA på tavlan. regler: se sid 24 Tiger-boken (blå).)

$\epsilon - closure\{s_1, \cdots s_n\}$ = alla tillstånd som kan nås från $s_1, \cdots s_n$ via ϵ-övergångar. (förkortas clos nedan).

```
e-closure{1} = {1}
DFAedge{1} 'i' = clos{2,4} = {2,4,5,6}
DFAedge{1} (c /= 'i') = clos{4} = {4,6,5}

DFAedge{2,4,5,6} 'f' = clos{3,6} = {3,5,6}
DFAedge{2,4,5,6} (c/='f') = clos{6} = {5,6}

DFAedge{3,5,6} (a-z0-9) = clos{6} } {5,6}

DFAedge{5,6} (a-z0-9) = clos{6} = {5,6}

DFAedge{4,5,6} (a-z0-9) = clos{6} = {5,6} .
```

I tillstånd {3,5,6} är både IF och ID sluttillstånd (IF i 3, ID i 6). Enligt regel-prioritetsregeln tas då IF, eftersom den står före ID.