Lexical analysis, scanners

(NB. Pages 41 - 49 are for those who need to refresh their knowledge of DFAs and NFAs. These are not presented during the lectures)

Function

1. Read the input stream (sequence of characters), group the characters into primitives (tokens). Returns token as *<type, value>*.
2. Throw out certain sequences of characters (blanks, comments, etc.).
3. Build the symbol table, string table, constant table, etc.
4. Generate error messages.
5. Convert, for example, string → integer.

Tokens are described using regular expressions.

Construction of a scanner

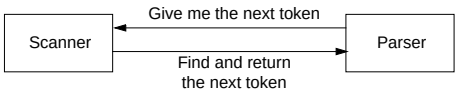
Tools: state automata and transition diagrams.

Regular expressions enable the automatic construction of scanners.

Scanner generator (e.g. *Lex*)**:**

In: Regular expression.
Out: Scanner.

Environment:



Finite state automata and diagrams

(*Finite automaton*)

Assume:

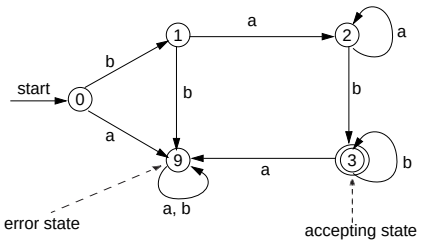
regular expression RU = $\mathtt{ba^+b^+ = baa\ ...\ abb\ ...\ b}$

$\mathtt{L(RU) = \{\ ba^nb^m \mid n, m \geq 1\ \}}$

Recognizer

A program which takes a string x and answers yes/no depending on whether x is included in the language.

The first step in constructing a recognizer for the language L(RU) is to draw a state diagram (transition diagram).



state diagram (DFA) for $\mathtt{ba^nb^m}$

Method of operation:

1. Start in the starting node 0.
2. Repeat until there is no more input:
   a) Read input.
   b) Follow a suitable edge.
3. When there is no more input:
   Check whether we are in a final state. In this case accept the string.

There is an error in the input if there is no suitable edge to follow.

Take this measure:
Add one or several error nodes.

Example of input: **baab**

Step	Current state	Input
1	0	baab
2	1	aab
3	2	ab
4	2	b
5	3	ε

Then *accept* because there is no more input and state 3 is an accepting state.

State diagrams are represented by **transition tables**:

Transition table				
State	Accept	Found	Next state	
			a	**b**
0	**no**	ε	**9**	**1**
1	**no**	**b**	**2**	**9**
2	**no**	**ba$^+$**	**2**	**3**
3	**yes**	**ba$^+$b$^+$**	**9**	**3**
9	**no**	**?**		**9**

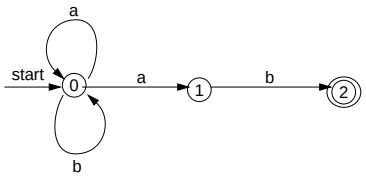
(Suitable for computer representation).

The previous graph is a DFA (*Deterministic Finite Automaton*).

It is deterministic because at each step there is exactly one state to go to and there is no transition marked ''ε''.

A regular expression denotes a regular set and corresponds to an NFA *(Nondeterministic Finite Automaton).*

Example: NFA for **(b|a)*ab**



state diagram for **(b|a)*ab**

state	a	b	Accept
0	{0,1}	{0}	no
1		{2}	no
2			yes

Transition table for **(b|a)*ab**

It requires more calculations to simulate an NFA with a computer program, e.g. for input **ab**.

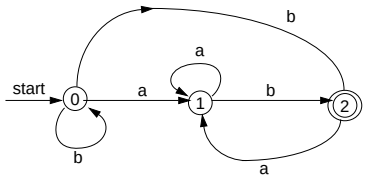
Theorem

Any NFA can be transformed to a corresponding DFA.

When generating a scanner automatically, the following is done:

- regular expression → NFA.
- NFA → DFA.
- DFA → minimal DFA.
- DFA → corresponding program code or table.

DFA for **(b|a)*ab**



Example: Design a DFA which can recognize what can follow **INTEGER** in a FORTRAN declaration:

```
INTEGER A
INTEGER X, Y, I(3)
INTEGER B(3, J, K), L
```

Alphabet Σ = { **v, c, '(', ')', ','** }
(**v** = variable, **c** = integer constant)

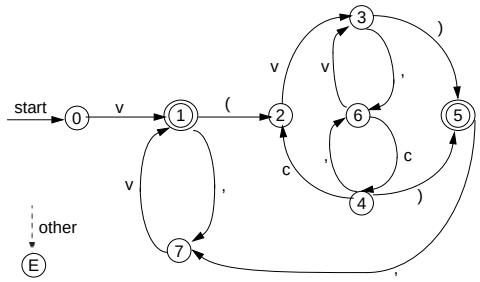
singlev = **v | v'('(v|c)(','(v|c))*')'**

singlev denotes a single variable or array.

r.e. = *singlev* (',' *singlev)**

state

0 = start
1 = variable is found
2 = variable is found, then **'('**
3 = variable is found, then **'('**, then variable which states dimension
4 = variable is found, then **'('**, then integer constant which states dimension
5 = **')'** after 3, 4 marks end of dimension
6 = **','** after 3, 4 marks dimension separator
7 = **','** after 1, 5 marks variable separator
E = error state



NB. Not a minimal DFA.
Can be optimized by merging states.
Study the transition table:

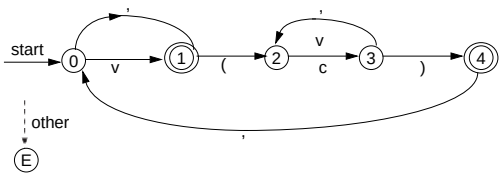
Transition table						
state	v	c	(	)	,	Accept
0	1	E	E	E	E	no
1	E	E	2	E	7	yes
2	3	4	E	E	E	no
3	E	E	E	5	6	no
4	E	E	E	5	6	no
5	E	E	E	E	7	yes
6	3	4	E	E	E	no
7	1	E	E	E	E	no
E	E	E	E	E	E	no

Merge all similar states (similar rows). Note that you cannot merge two states if only one of them is an accepting state. (0 = 7, 2 = 6, 3 = 4)
Re-number the states,

which gives us:

Optimized transition table						
state	v	c	(	)	,	Accept
0	1	E	E	E	E	no
1	E	E	2	E	0	yes
2	3	3	E	E	E	no
3	E	E	E	4	2	no
4	E	E	E	E	0	yes
E	E	E	E	E	E	no

This corresponds to a minimal DFA below:



How is a scanner programmed?

1. Describe tokens with regular expressions.
2. Draw transition diagrams.
3. Code the diagram as table/program.

Example. Write a scanner for the following tokens.

Several categories of tokens:

keyword = **`BEGIN`** | **`END`**
id = letter (letter | digit)*
integer = digit+
op = `+` | `-` | `*` | `/` | `//` | ↑ | `=` | `:=`

Simplification:

Assume that there is a blank character after each token.

This simplification can easily be removed!

The scanner represents tokens as tuples:

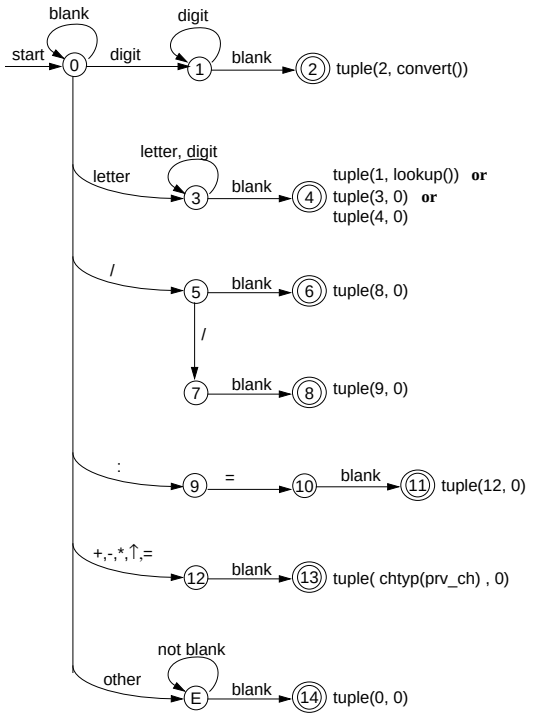
Tuple type	< Typecode, value >
undef.	`< 0,0 >`
id	`< 1,` table-pointer `>`
integer	`< 2,` value `>`
`BEGIN`	`< 3,0 >`
`END`	`< 4,0 >`
`+`	`< 5,0 >`
`-`	`< 6,0 >`
`*`	`< 7,0 >`
`/`	`< 8,0 >`
`//`	`< 9,0 >`
↑	`< 10,0 >`
`=`	`< 11,0 >`
`:=`	`< 12,0 >`

Practical trick for 1-character tokens:

Initialize a vector **`chtyp`**

A	B	...	+	-	*	↑	=	...
0	0	...	5	6	7	10	11	...

Draw the transition diagram:



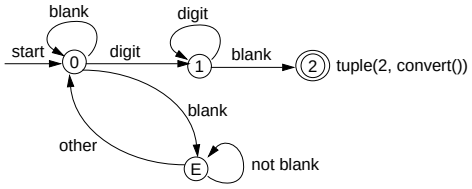
Comments:

- **`convert()`** converts text to integers.
- **`lookup()`** returns index to symbol table.
- **`BEGIN`**, **`END`** dealt with by putting them in the symbol table from the beginning. When they are found, return 3 or 4 instead of 1.
- **`ch`** always contains the next character
- **`prv_ch`** always contains the next to the last character.
- Automatic transition to state 0 after each recognized token (even after state 14).

Coding

1. Transpose the diagram to a transition table, perform simple interpretation of the table.

Example: Transition- diagram and -table for integers:



state	digit	Blank	Other	Accept
0	**1**	**0**	**E**	**0**
1	**1**	**2**	**E**	**0**
2	**-**	**-**	**-**	**1**
E	**E**	**0**	**E**	**0**

Different variants of coding the table

a) Interpreting the table

state	digit	Blank	Other	Accept
0	**1**	**0**	**E**	**0**
1	**1**	**2**	**E**	**0**
2	**-**	**-**	**-**	**1**
E	**E**	**0**	**E**	**0**

```
...
Read in or initiate table table[state,ch];
ch := getchar;
state := 0;      (* starte state *)
while not eof do
begin
 oldstate := state;
 state := table[state, ch];
 accumulate(ch);
 if state = error_state then
     error_handling(oldstate, ch, ...)
 else if accept[state] then
         perform_accept(state);
         (* return tuple and state := 0 *)
 ch := getchar;
end;
```

b) **GOTO-representation of table**

state	digit	Blank	Other	Accept
0	1	0	E	0
1	1	2	E	0
2	-	-	-	1
E	E	0	E	0

```
state0:
   ch := getchar;
   if ch = digit then goto state1;
   if ch = ' ' then goto state0;
   goto stateE;    (* in other cases *)

state1:
   ...
```

c) using a CASE statement

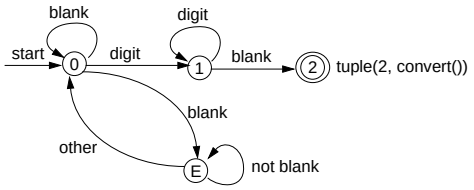
```
case state of
 0:
    case ch of
     '0'..'9': state := 1;
     ' ':      state := 0;
     others:   state := E;
    end;
 1:
    ...
```


2. Direct coding of diagrams (not via table)

Global variables:
- **toktyp =** current symbol class
- **value =** value
- **ch =** current character
- **eofile =** end-of-file flag
- **chtyp =** vector for 1-character tokens
- **symtab =** symbol table

Routines:
- **getchar;**
- **skip_blanks;**
- **accumulate(ch);**
- **lookup(id);**
- **letter(ch);**
- **digit(ch);**

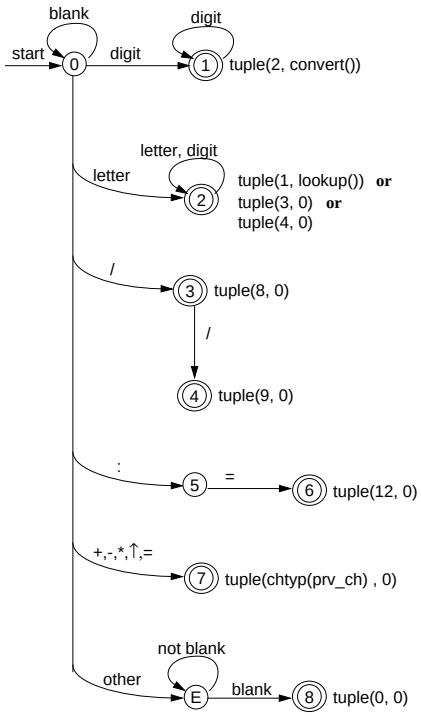
Initiate:
- **eofile := false;**
- initiate **chtyp** according to the previous description;
- initiate the symbol table with reserved words;



Incomplete suggested program for the scanner

```
ch := getchar;
val := 0;
while not eofile do
begin
   skip_blanks;
   case ch of
    'A'..'Z':
       ch := getchar;
       while letter(ch) or digit(ch) do
          ch := getchar;
       if ch = ' ' then
        lookup(accumulated string)
       else error(...);
    '0'..'9':
       val := ord(ch) - ord('0');
       ch := getchar;
       while digit(ch) do begin
          val:=val*10+ord(ch)-ord('0');
          ch := getchar;
       end;
       if ch = ' ' then toktyp := 2
       else error(...);
    ...
    others:
       toktyp := chtyp[ch];
       if toktyp = 0 then
           error(...);
   end (* case *);
end (* while *);
```


Simplification removed (i.e. not necessarily a concluding blank character):



Problem:
Lookahead is sometimes needed to determine symbol type.

Example: in FORTRAN

`DO 10 I = 1.25` is an assignment, but

`DO 10 I = 1,25` is a for-statement.

It is '`.`' or '`,`' which determines whether the scanner returns `DO10I` or `DO`

Example: in Pascal

```
715..816
```