Lexical Analysis (Scanning)

José Miguel Rivero

rivero@cs.upc.edu

Barcelona School of Informatics (FIB)

Universitat Politècnica de Catalunya BarcelonaTech (UPC)

Credits

Some of the material in these slides has been extracted from:

- the one elaborated by Prof. Stephen A. Edwards (University of Columbia) for the course COMS W4115 (Programming Languages and Translators)
- the ones elaborated by Profs. Jordi Cortadella, Guillem Godoy and Robert Nieuwenhuis (Barcelona Tech (UPC)) for the course Compilers (Barcelona School of Informatics)

Summary

- Objectives of Lexical Analysis
- Scanning in Compilers / Interpreters
- Regular Expressions. Applications
- The Basic Problem: $w \in \mathcal{L}(er)$?
  - Nondeterministic Automata NFA(re)
  - Deterministic Automata DFA(re)
  - Comparing Both Approaches
- The Problem of Lexical Analysis
- Lexical Errors. Recovery
- Automatic Generation of Scanners: *ANTLR*, *flex*, ...

FIB

Objective. Tokens

- Objective:
  split the sequence of characters of the source program into a sequence of lexical components (*tokens*)
- Tokens to be recognized and **be sent** to the parser:
  - language keywords (`while`, `vars`, `write`)
  - operators (`+`, `/`, `<=`, `OR`, `:=`)
  - punctuation symbols (parenthesis, comma, semicolon)
  - identifiers (`numels`), integer values (`834`), strings (`"Hello world!"`), floats (`3.04E-3`)
  - ...

FIB

Other Lexical Components

- Tokens to be recognized but **without interest** for later phases:
  - separators: blanks, tabs
  - comments: `/* ... */` in C, `# ...` in Perl
  - newlines. To localize syntactical errors

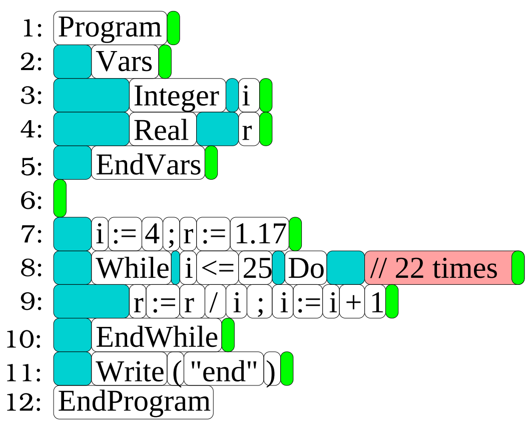
Token Attributes

- for all of them: the position
- for identifiers, numerical values, strings: the corresponding text (“v0”, “54.7”)

Example

- Source program:

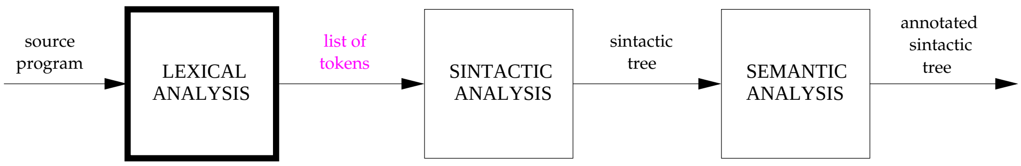
```
Program
   Vars
      Integer  i
      Real     r
   EndVars

   i := 4 ; r := 1.17
   While  i <= 25  Do   // 22 times
      r := r  /  i  ;  i := i + 1
   EndWhile
   Write ( "end" )
EndProgram
```



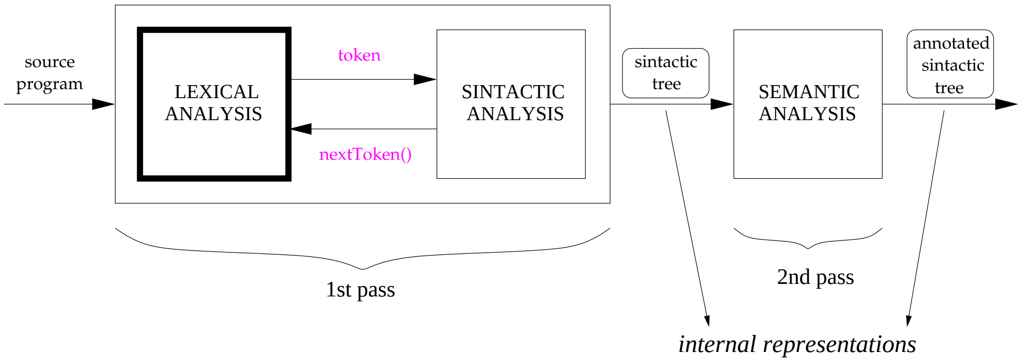
- Sequence of tokens: PROGRAM VARS INTEGER IDENT("i") REAL IDENT("r") ENDVARS IDENT("i") ASSIG INTCONST("4") SEMI IDENT("r") ASSIG REALCONST("1.17") WHILE IDENT("i") LESS INTCONST("25") DO IDENT("r") ASSIG IDENT("r") REALDIV IDENT("i") SEMI IDENT("i") ASSIG IDENT("i") PLUS INTCONST("1") ENDWHILE WRITE LEFTPAR STRINGCONST("end") RIGHTPAR ENDPROGRAM

Scanning in Compilers / Interpreters

- Conceptual structure



- Usual structure



Motivation

Why a specific phase for the lexical analysis?

- Conceptually is a specialized task: filter and break the input in those items interesting for the next phase, the syntactical analysis
- Applied techniques are
  - simple and efficient: *"Not use a sledgehammer to crack a nut"*
  - flexible (lexical changes can be easily resolved)
  - portable and general
- These techniques are applied in many other applications

Some Applications

- Information retrieval queries
- Genetic problems
- Syntax-driven text editors
- Operating systems (shell script languages, *grep*)
  Example (in *unix*): `% rm prog*.[ch]`
- Pattern/action programming languages: (*awk*)
- Analysis of digital circuits
- State controllers of video games
- ...

Regular Expressions

Lexical components of a language are specified through regular expressions over an alphabet Σ.

Formation rules:

- $re = \epsilon$ is a regular expression
- $re = a$ is a regular expression for all $a \in \Sigma$
- if re_1 and re_2 are regular expressions,
  $re = re_1 | re_2$ is a regular expression
- if re_1 and re_2 are regular expressions,
  $re = re_1 re_2$ is a regular expression
- if re_1 is a regular expression $re = re_1^*$, $re = re_1^+$ and
  $re = re_1?$ $(re_1 | \epsilon)$ are regular expressions
- if re_1 is a regular expression, $re = (re_1)$ is a regular expression

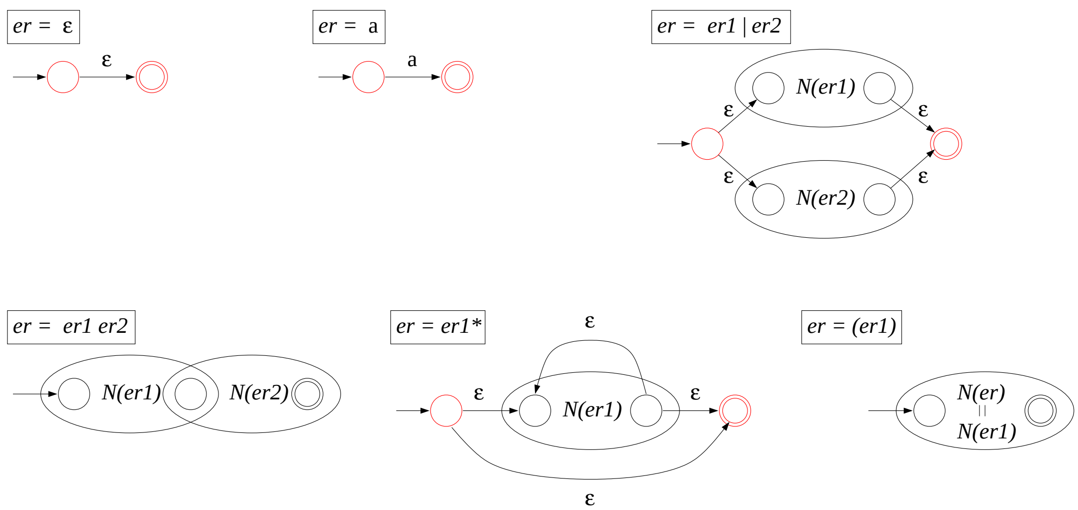
Expressive Power

- The set of well-balanced expressions, for example $\{\, a^n b^n \mid n > 0 \,\}$, cannot be accepted by a finite automata: *“finite automata cannot count”*
- Neither can be accepted the words of the language $\{na^n \mid n \geq 0\} = \{\, 0,\ 1a,\ 2aa,\ 3aaa,\ \ldots\}$
- The language of repeated strings $\{\, w\,c\,w \mid w \in (a|b)^* \,\}$ cannot be described by a regular expression, nor even by a context-free grammar.

NFA(re) Construction

- **Thompson’s algorithm**: transform a regular expression er into a nondeterministic automata $N(er)$.

  Given the regular expressions ϵ, a, $re_1 \,|\, re_2$, $re_1\, re_2$, re_1^* and (re_1), and the automata $N(re_1)$ and $N(re_2)$:

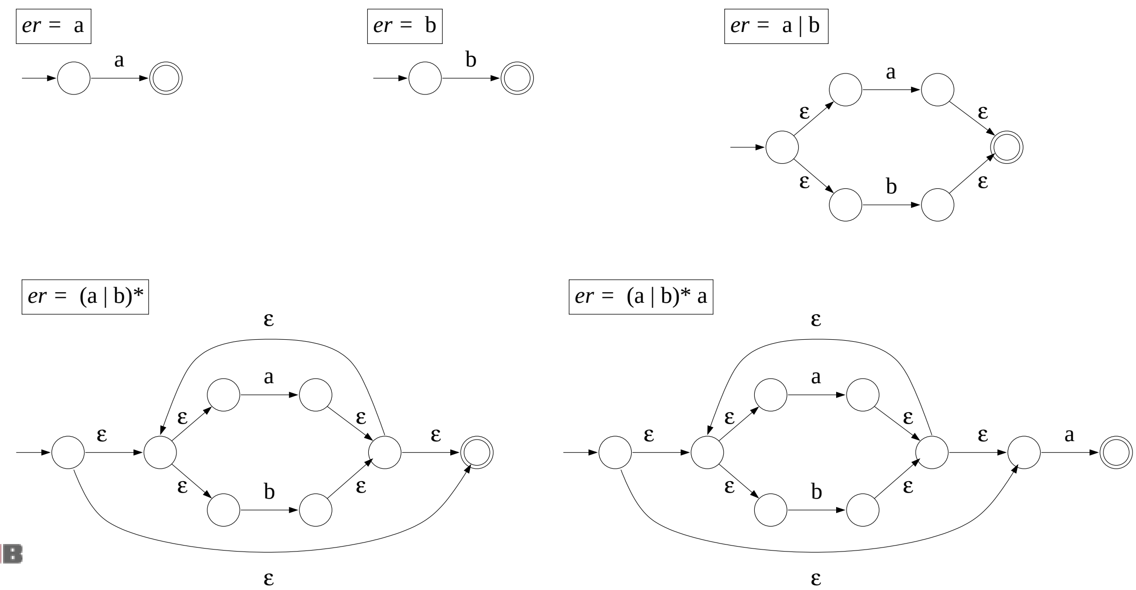


NFA(re) Construction

- Construction invariant: every NFA have an initial state without input edges, and only one final state without output edges
- The number of states of NFA(re) $\leq 2|re|$, because at most 2 new states are added at each construction step
- There are at most 2 output edges (2 transitions) for each automata's state. Therefore, we obtain a compact representation of the automata

Example 1

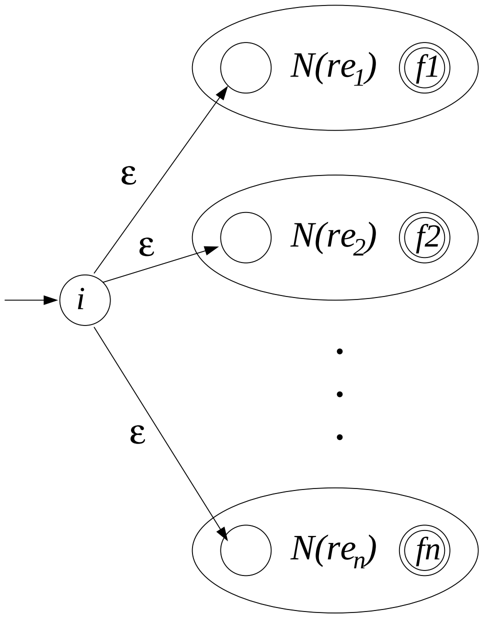
Nondeterministic finite automata for the regular expression
$re = (a|b)^*abb$.
These are the first steps of the $NFA(er)$ construction:



Example 2

Combination of NFA's for the disjunction of a set of regular expressions re_i's (*similar* to lexical analysis)

$\boxed{re = re_1 \mid re_2 \mid \ldots \mid re_n}$



Decision Algorithm for $w \in$ NFA(re)

First we define two auxiliary functions:

$\epsilon\text{-}closure(S)$ is the set of states accessible from states in S with zero o more ϵ-transitions.

$move(S, a)$ is the set of states accessible from states in S with a transition labelled with a.

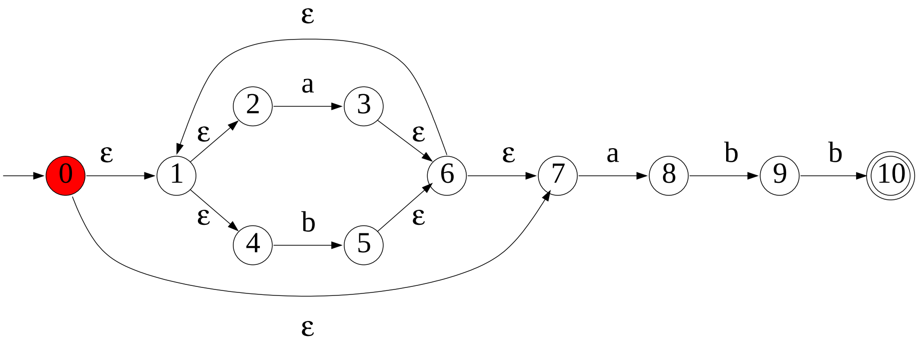
Algorithm to decide if $w \in$NFA(re):

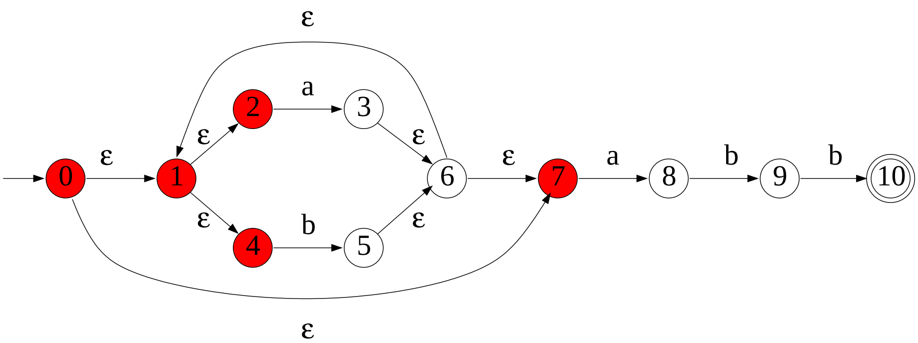
$Pre:$ s_0 is the initial state of the automata NFA
F is the set of final states of NFA
eof is the symbol ending w

$$
\begin{array}{l}
S := \epsilon\text{-}closure(\{s_0\});\\
a := NextSymbol(\,);\\
while\ \ a\ != eof\ \ do\\
\quad S := \epsilon\text{-}closure(move(S, a));\\
\quad a := NextSymbol(\,);\\
endwhile
\end{array}
$$

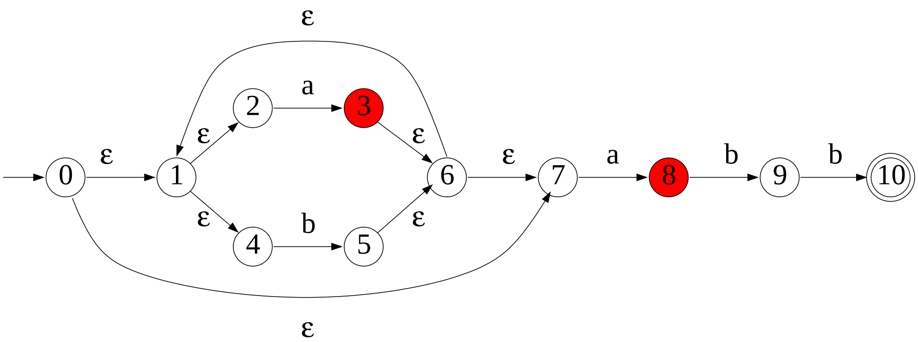
$Post: NFA$ accepts w iff $S \cap F \neq \emptyset$

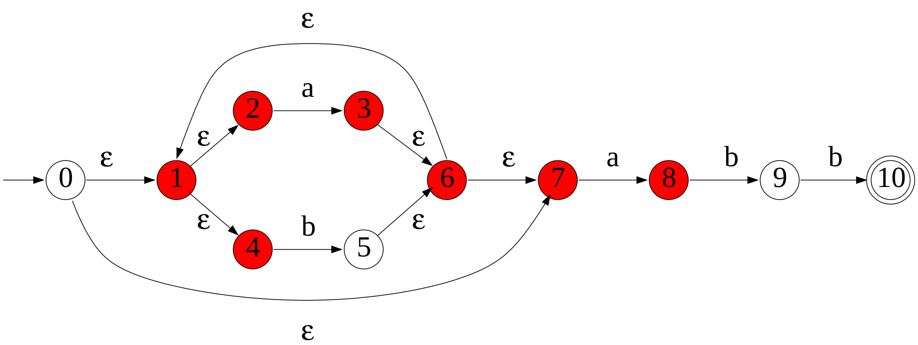
Simulating the input: $\cdot\, aabb$



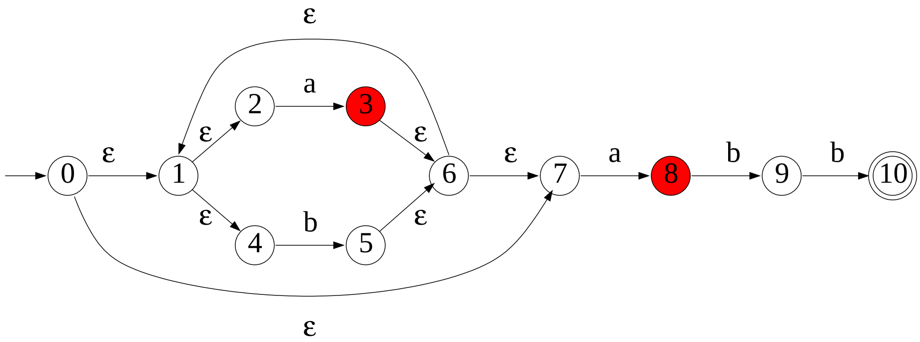


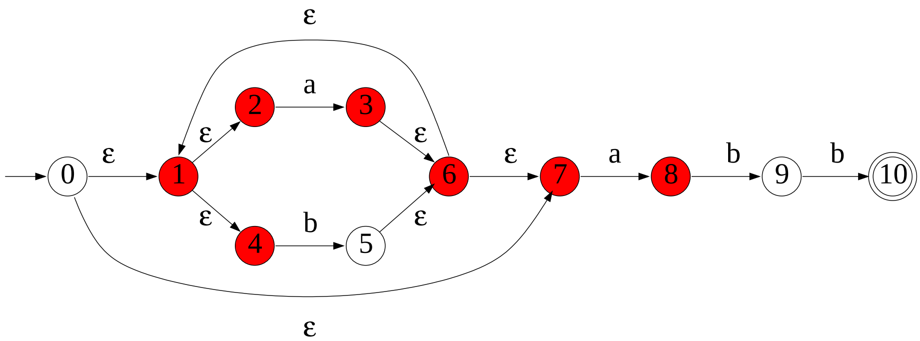
Simulating the input: $a \cdot abb$



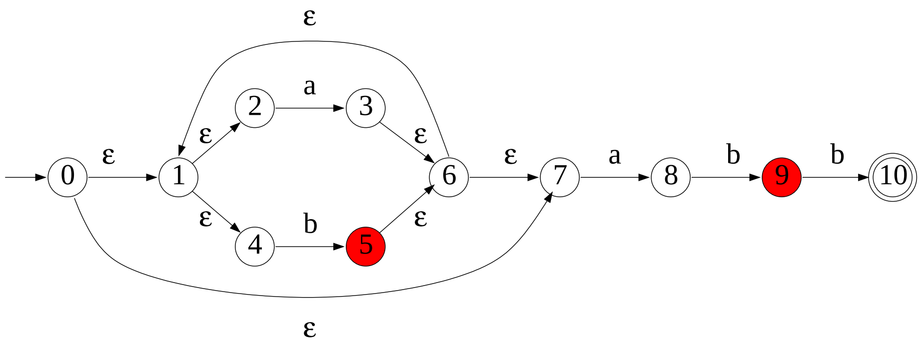


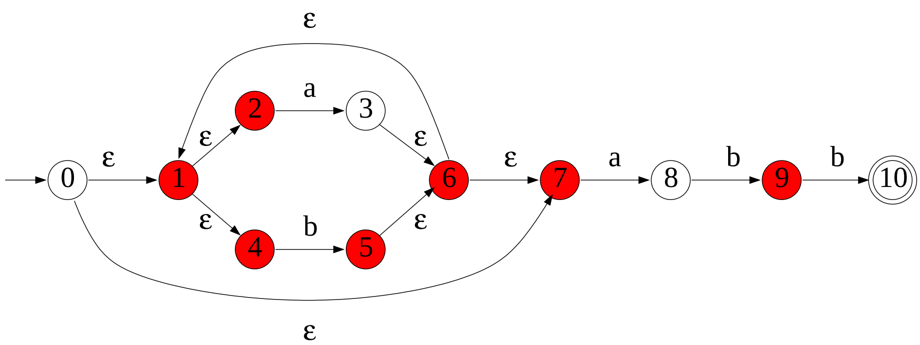
Simulating the input: $aa \cdot bb$



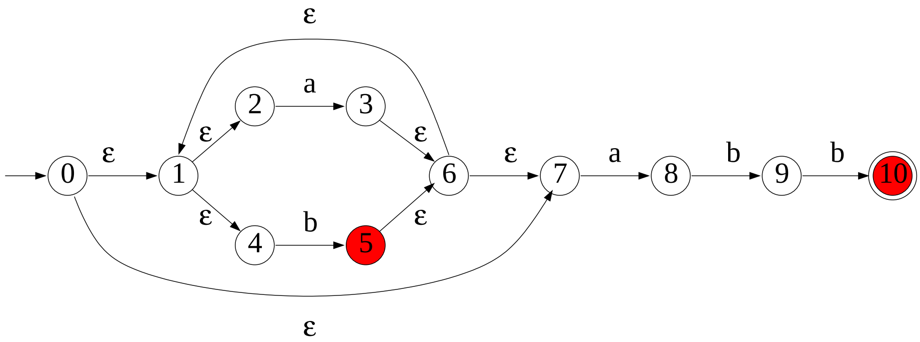


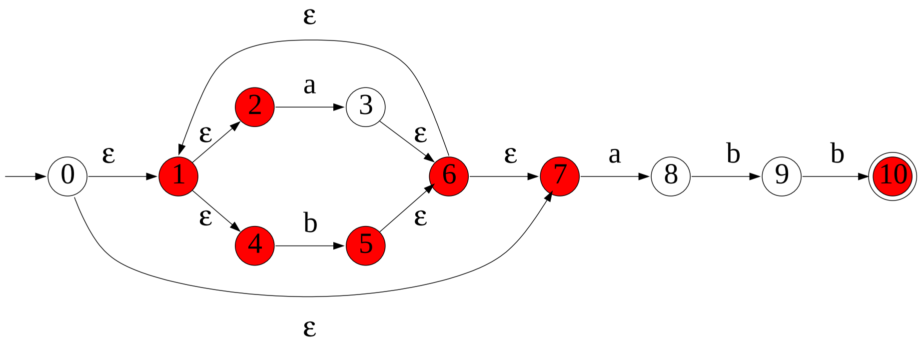
Simulating the input: $aab \cdot b$





Simulating the input: $aabb\ \cdot$





Algorithm Costs

- Temporal cost:
  $$O(|re| \cdot |w|)$$
- Spatial cost (size of NFA's transition table):
  $$O(|re|)$$

DFA(re) Construction

- Determination algorithm. Example
- DFA spatial cost
- Minimization algorithm. Example
- Decision algorithm for $w \in$ DFA(er)
- Compression techniques

Determination Algorithm

- Deterministic finite automata: no ϵ-transition nor any state with more than one edge for the same symbol $a \in \Sigma$.
- Computing subsets of states. Each *possible* subset of states in the NFA will correspond to one state in the DFA. Transitions between these states will be computed
- Algorithm:
  - *Dstate* (the set of DFA states) and *Dtran* (the DFA transition table) will be computed.
  - A state in *Dstate* will be *marked* when all their transitions in *Dtran* have been defined

Determination Algorithm

Pre: s_0 is the initial state of NFA

F is the set of final states of NFA

ϵ-*closure*$(\{s_0\})$ is the only state in $Dstate$ and is not marked

while exist a state S not marked in $Dstate$ *do*

mark S

foreach input symbol $a \in \Sigma$ *do*

$S' := \epsilon\text{–}closure(move(S, a));$

if $S' \notin Dstate$ *then*

add S' (without mark) to $Dstate$

endif

$Dtran[S, a] := S';$

endfor

endwhile

Post: The initial state of DFA is ϵ-*closure*$(\{s_0\})$

Final states of DFA are those (sets of)

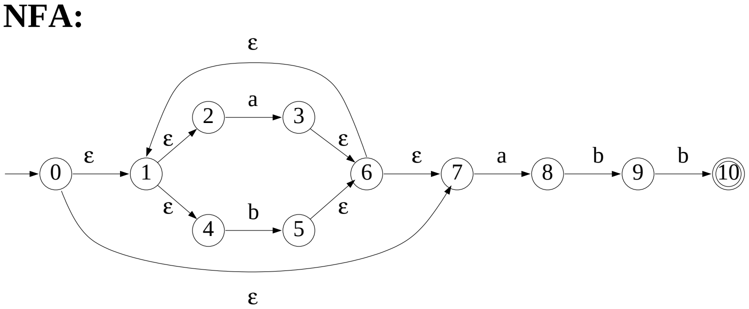
states containing at least one state of F

Example

Compute the deterministic FA for the regular expression

$$re = (a|b)^*abb$$

NFA:



$$\begin{aligned}
\epsilon\text{-}closure(\{0\}) &= \{0,\ 1,\ 2,\ 4,\ 7\} = A \\
\epsilon\text{-}closure(move(A, a)) &= \epsilon\text{-}closure(\{3,\ 8\}) \\
&= \{1,\ 2,\ 3,\ 4,\ 6,\ 7,\ 8\} = B
\end{aligned}$$

$Dtran[A,\ a] = B$

$$\begin{aligned}
\epsilon\text{-}closure(move(A, b)) &= \epsilon\text{-}closure(\{5\}) \\
&= \{1,\ 2,\ 4,\ 5,\ 6,\ 7\} = C
\end{aligned}$$

$Dtran[A,\ b] = C$

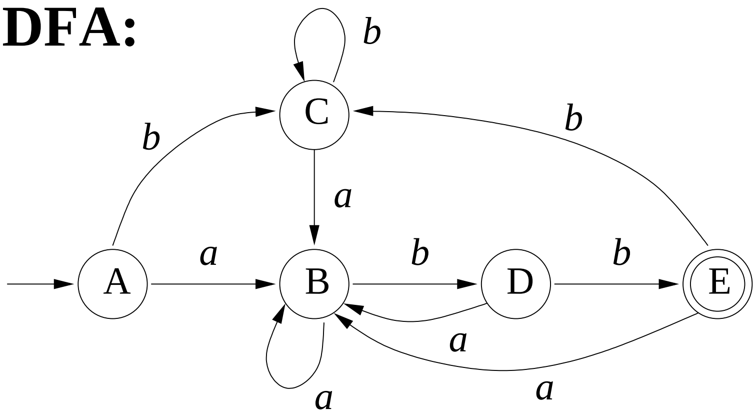
...

Example

Dtran:

	symbol	
state	*a*	*b*
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

DFA:



DFA(re) Spatial Cost

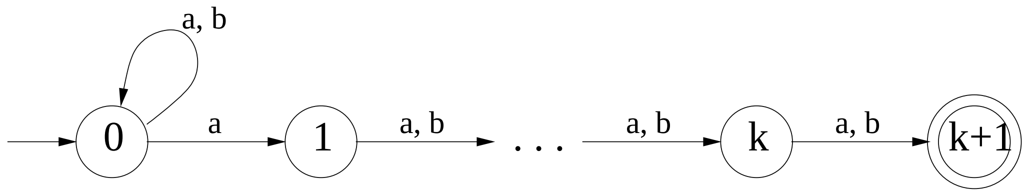
The spatial cost (number of states in $Dtran$) may be exponential wrt. the length of re:

The number of different subsets of a set of N *elements is* 2^N

Example: Given the regular expression $(a|b)^*a(a|b)^k$, the automata NFA will be constructed in the following way:

- An initial state 0 with edges labelled with a and b towards itself, and an edge labelled with a towards state 1
- Transitions from state i labelled with a and b towards state i+1, for $i \in [1..k]$
- State k+1 is the final state

DFA(re) Spatial Cost



The size of the corresponding DFA is exponential in k because it needs to remember $k+1$ bits (the latest $k+1$ symbols that have been read)

With $k = 3$:

$\underline{a}bba$ (final state) $\longrightarrow^a$ $\underline{b}baa$ (non-final state)

$\underline{b}aba$ (non-final state) $\longrightarrow^b$ $\underline{a}bab$ (final state)

FIB

DFA Minimization Algorithm

Compute successive partitions of the set of states.

$Pre:$ S is the set of DFA states

s_0 is the DFA initial state

F is the set of DFA final states

$Post:$ DFA' accepts the same language than DFA

having the minimum number of states

FIB

DFA Minimization Algorithm

Compute successive partitions of the set of states.

```
initial partition Π = Π_new with two grups :
          final states F  and non-final states S \ F
repeat
    Π := Π_new
    for each grup G of Π do
        1.  divide G in subgrups s.t. two states s and t
            of G leave in the same subgrup iff for all
            symbol a ∈ Σ, s and t have transitions
            towards states in the same subgrup of Π.
        2.  replace G in Π_new by the set of formed subgrups
    endfor
until Π_new = Π
```


DFA Minimization Algorithm

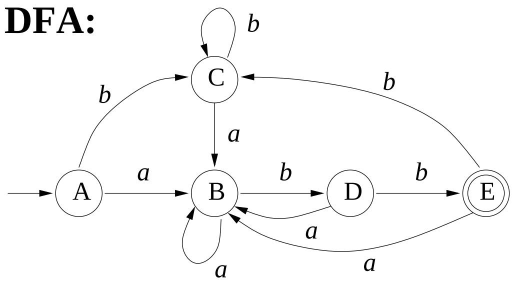
Now build the automata DFA':

1. Its states are defined choosing a representative for each group
2. Transitions in DFA' will correspond to the transitions between the representative states in the DFA
3. The initial state of DFA' will be the representative of the group containing s_0
4. The final states will be those having representatives in F

Example

Minimization of the DFA that recognizes $(a|b)^*abb$

DFA:



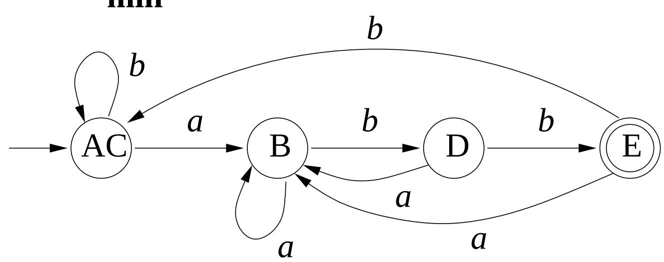
Comments	Partitions
non-final / final states	$(ABCD)$ | (E)
$A, B, C \rightarrow^b (ABCD)$ but $D \rightarrow^b (E)$	(ABC) | (D) | (E)
$A,\ C \rightarrow^b (ABC)$ but $B \rightarrow^b (D)$	(AC) | (B) | (D) | (E)
final partition	

Example

Dtran:

	symbol	
state	*a*	*b*
AC	B	AC
B	B	D
D	B	E
E	B	AC

DFA$_{\mathbf{min}}$:



Another way to construct DFA(re)

- Avoids determinating the NFA(re), and applying subsequently the minimization algorithm.
  Carry out these two steps in one
- Not always obtain the minimum DFA(re) but is a good technique in most cases
- Comment very briefly ...

Decision Algorithm for $w \in$ DFA(re)

$Pre:$ s_0 is the initial state of the automata DFA
F is the set of final states of DFA
eof is the ending symbol of w

$s := s_0;$
$a := NextSymbol(\,);$
$while\ \ a\ \ != \ eof\ \ do$
$\quad s := Dtran[s,\, a];$
$\quad a := NextSymbol(\,);$
$endwhile$

$Post:$ DFA accepts w iff $s \in F$

- Temporal cost: $O(|w|)$
- Spatial cost (size of $Dtran$):
  $O($ (number of states of the DFA) $*$ (number of symbols of Σ) $)\ =\ O(2^{|re|})$

Compression Techniques

- Different implementations for the DFA transition function: the most direct using a transition table.
- $size(\text{Dtran}) = \#states \cdot |\Sigma|$
- Usually:
  1. the number of states is very high, and
  2. for each state: most of transitions are undefined, or go to the same state
- So this huge table may be quite empty (*sparse table*)

FIB

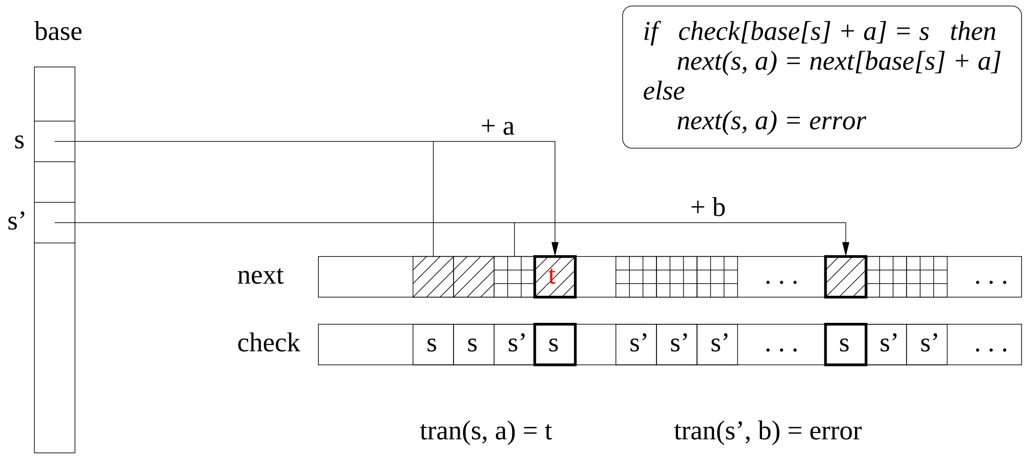
Compression Techniques

- One dimensional vector of states.
  For each state we have the list of defined transitions plus the default transition in case of error.
  Very easy but make worse the time of compute a transition
- Other techniques seek to exploit, for each state, contiguous empty squares before the first and after the last symbol with transition.
  - Use several additional tables
  - Improves the time to compute a transition
  - Wasted space is much lower

FIB

Compression Techniques

- Two or more rows can be overlapped when transitions defined in both don't match.
  This technique is shown in the following figure:



Comparing Both Approaches

Summing up costs:

Automata	Temporal cost	Spatial cost
NFA	$O(\|re\| \cdot \|w\|)$	$O(\|re\|)$
DFA	$O(\|w\|)$	$O(2^{\|re\|})$

In general, when both methods are feasible (the DFA spatial cost is reasonable) the following could be concluded:

NFA is suitable when $|re| \downarrow\downarrow$

DFA is suitable when $|re| \uparrow\uparrow$ or $|w| \uparrow\uparrow$

Lazy Finite Automata

- Combines: space requirements of NFA with advantage in time of DFA
- Works like an indeterministic automaton, computing only the subsets of states that are needed.
  These subsets (and their transitions) are stored in a *cache* so it is not required to recompute them again.
- To sum up:
  - Lower requirements of space:
    size of NFA transition table ($O(|re|)$) $+$ size of *cache*
  - Transitions for non used states are not computed
  - Nearly as fast as DFA

FIB

The Problem of Lexical Analysis

- Problem description
- Criteria to remove ambiguities
- Examples
- An algorithm for lexical analysis
- Lexical errors
- Be careful with the language!

FIB

Problem Description (v0)

Given a list of regular expressions $re_1, \ldots, re_n$ describing the n different *tokens* that can be recognized, and a word w (the source program), it must be found a partition $v_1 v_2 \cdots v_k$ of w such that each subword v_i is in the language of some re_j.

Example 1:

$$er_1 = bca$$
$$er_2 = a^*bc$$
$$w = bcabc$$

solution #1: $v_1 = bca \in \mathcal{L}(er_1)$ and $v_2 = bc \in \mathcal{L}(er_2)$

solution #2: $v_1 = bc \in \mathcal{L}(er_2)$ and $v_2 = abc \in \mathcal{L}(er_2)$

so more precisely...

Problem Description (v1)

Given a list of regular expressions $re_1, \ldots, re_n$ and a word w, it must be found a partition $v_1 v_2 \cdots v_k$ of w such that each word v_i is the longest successive prefix in the language of some re_j.

Example 1:

$$er_1 = bca$$
$$er_2 = a^*bc$$
$$w = bcabc$$

solution: $v_1 = bca \in \mathcal{L}(er_1)$ and $v_2 = bc \in \mathcal{L}(er_2)$

Problem Description (v1)

Given a list of regular expressions $re_1, \ldots, re_n$ and a word w, it must be found a partition $v_1 v_2 \cdots v_k$ of w such that each word v_i is the longest successive prefix in the language of some re_j.

Example 2:

$$er_1 = a(b|c)$$
$$er_2 = a^*c$$
$$er_3 = b \qquad\qquad w = acb$$

solution #1: $v_1 = ac \in \mathcal{L}(er_1)$ and $v_2 = b \in \mathcal{L}(er_3)$

solution #2: $v_1 = ac \in \mathcal{L}(er_2)$ and $v_2 = b \in \mathcal{L}(er_3)$

so even more precisely...

Problem Description (v2)

Given a list of regular expressions $re_1, \ldots, re_n$ and a word w, it must be found a partition $v_1 v_2 \cdots v_k$ of w such that each word v_i is the longest successive prefix in the language of some re_j.
If some longest prefix v_i is in the language of more than one *token*, the regular expression with the lowest index will be selected.

Example 2:

$$er_1 = a(b|c)$$
$$er_2 = a^*c$$
$$er_3 = b$$
$$w = acb$$

solution: $v_1 = ac \in \mathcal{L}(er_1)$ and $v_2 = b \in \mathcal{L}(er_3)$

Problem Description (v2)

Given a list of regular expressions $re_1, \ldots, re_n$ and a word w, it must be found a partition $v_1 v_2 \cdots v_k$ of w such that each word v_i is the longest successive prefix in the language of some re_j.
If some longest prefix v_i is in the language of more than one *token*, the regular expression with the lowest index will be selected.

These restrictions may make impossible to find a solution even when a partition exists:

Example 3:

$$er_1 = a^*b$$
$$er_2 = aa$$
$$er_3 = bc \qquad\qquad w = aabc$$

Problem Description

Given a list of regular expressions $re_1, \ldots, re_n$ and a word $w = v\,w'$, it must be found the longest prefix v of w s.t. $v \in \mathcal{L}(re_j)$.
If $v \in \mathcal{L}(re_j)$ for more than one re_j, the regular expression with the lowest index j will be selected.

The lexical analyzer, the function `nextToken()`, returns both the prefix v and the index j indicating the recognized token.

The next call to `nextToken()` makes the same with the remaining input w'.

Criteria to Remove Ambiguities

- Recognize always the longest prefix
- Specify the regular expressions corresponding to keywords before (lowest i) than the identifiers: any keyword is also a word in the language of the identifiers, but must be recognized as keyword.
- Example of some tokens specified in *PCCTS*:

```
#token   PROGRAM       "PROGRAM"
#token   VARS          "VARS"
         ...
#token   COMMA         ","
         ...
#token   INT_CONST     "[0-9]+"
#token   IDENT         "[A-Za-z][A-Za-z0-9]*"
         ...
```


Some Examples

- With the input `"whilei>5 ..."` it will **not** be obtained the keyword `while` followed by the identifier `i`
- With `"if( ..."` it will **not** be recognized the identifier `"if"`. The keyword `"if"` takes precedence
- With `"ab24.8 ..."` it will **not** be recognized the identifier `"ab"` followed by the real `"24.8"` (unless the identifiers can only include alphabetical characters)
- With `"10..20 ..."` it will be obtained the integer `"10"` because, after trying to recognize a longer prefix (a real beginning with `"10."`), it fails in the second `'.'`. In successive calls, the tokens double-dot and another integer will retrieved.

  $\Rightarrow$ This introduce non-linearity! :-((

Non linearity

- Example:

$$er_1 = b^*a^*c$$
$$er_2 = a$$
$$er_3 = b \qquad w = \boxed{a}\boxed{a}\boxed{a}\boxed{a}\boxed{a}\boxed{a}\boxed{a}\boxed{a}\boxed{a}\boxed{a}\boxed{a}\boxed{a}\boxed{a}\boxed{a}\boxed{a}\boxed{a}\boxed{a}\boxed{a}\boxed{a}\boxed{b}$$

- Remember the position ending an accepted prefix, and the number of the DFA involved.
  If success in finding a longer one, update that information; otherwise, come back to the last successful point.
- To avoid non-linearity, and once some accepted prefix has been detected, it can be imposed that each new symbol also form a new longer recognized prefix

FIB

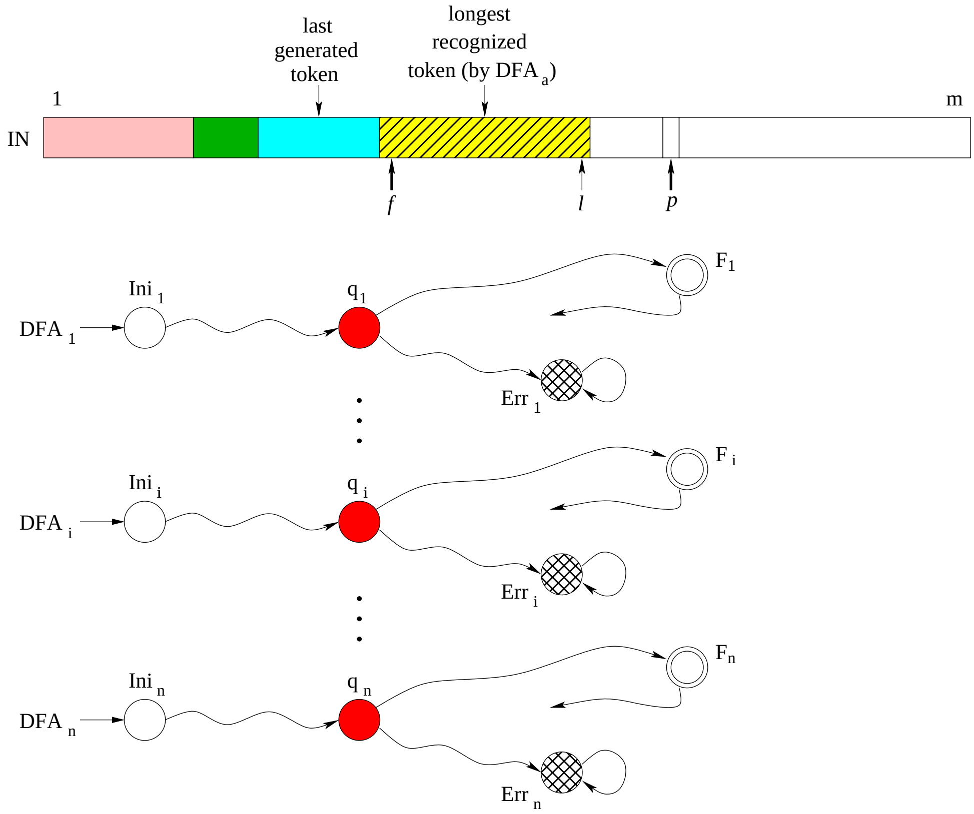
Lexical Analysis Algorithm

Exercise. Suppose that the symbols of w are in an array $IN[1..m]$, and the n DFA transition functions corresponding to the n tokens are δ_i (Dtran$_i$). Write an algorithm that partition the input —working over the DFA recognition algorithm for each token—, and obtain successive prefixes $v = IN[f] \cdots IN[l]$ matching some token a $(v \in \mathcal{L}(re_a))$.
The initial and final states of DFA$_i$ are Ini_i and the set F_i. When a DFA$_i$ does not define transition for state q and symbol $IN[p]$, then $\delta_i(q,\, IN[p])$ returns the value Err_i.
The partition algorithm successively returns the pair of indexes $\langle f, l \rangle$ and the number a of the DFA, such that:

- the word $IN[f] \cdots IN[l]$ is the longest prefix of $IN[f] \cdots IN[m]$ matching some re_i
- a is the minimal value of the i's for the re_i's that accept this longest prefix. If no prefix exists, a *lexical error* is generated

FIB

Lexical Analysis Algorithm



Lexical Analysis Algorithm (v0)

$p := f := 1;\ l := 0;$

$\forall i : 1 \le i \le n : q_i := \mathtt{Ini}_i;$ // *Initial states*

while $p \le m$ **do**

  $\forall i : 1 \le i \le n : q_i := \delta_i(q_i,\ \mathtt{IN}[p]);\ \ p := p + 1;$ // *State Transitions*

  if $\exists i : 1 \le i \le n : q_i \in \mathtt{F}_i$ **then** // *Some final state*

    $l := p - 1;\ \ a =$ *smallest* i *such that* $q_i \in F_i$;

  elseif $\forall i : 1 \le i \le n : q_i \in \mathtt{Err}_i$ **then** // *All* Err_i *states*

    if $l \ge f$ **then**

      Generate token of type a *with word* $\mathtt{IN}[f..l]$

      $p := f := l + 1;\ \ l := f - 1;$

      $\forall i : 1 \le i \le n : q_i := \mathtt{Ini}_i;$

    else

      Generate and Recover from a Lexical Error

    endif

  endif

endwhile

if $l < f$ **then** *Generate a Lexical error* **endif**

Lexical Analysis Algorithm

```
p := f := 1; l := 0;
∀i : 1 ≤ i ≤ n : q_i := Ini_i;                              // Initial states
while  f ≤ m  do
   while  p ≤ m  do
      ∀i : 1 ≤ i ≤ n : q_i := δ_i( q_i , IN[p] );  p :=p + 1;   // State Transitions
      if  ∃ i : 1 ≤ i ≤ n : q_i ∈ F_i  then                   // Some final state
         l := p − 1;  a = smallest  i  such that q_i ∈ F_i;
      elseif  ∀ i : 1 ≤ i ≤ n : q_i ∈ Err_i  then             // All Err_i states
         if  l ≥ f  then
            Generate token of type  a  with word  IN[f.. l]
            p  :=  f :=  l + 1;   l := f − 1;
            ∀i : 1 ≤ i ≤ n : q_i := Ini_i;
         else
            Generate and Recover from a Lexical Error
         endif
      endif
   endwhile
   if  l ≥ f  then
      Generate token of type  a  with word  IN[f.. l]
      p  :=  f :=  l + 1;   l := f − 1;
      ∀i : 1 ≤ i ≤ n : q_i := Ini_i;
   else
      Generate and Recover from a Lexical Error
   endif
endwhile
```


Lexical Analysis Algorithm

Recover from a Lexical Error :

$$p \ := \ f := \ f+1; \quad l := f-1;$$

$$\forall i \ : \ 1 \leq i \leq n \ : \ q_i := \texttt{Ini}_i;$$

Lexical Errors

How does a lexical error occur?

- Context: the lexical analyzer is looking for the longest prefix v of the input w , s.t. $v \in \mathcal{L}(re_i)$ for some i
- Suppose that on symbol a there is no defined transition from any of the current states q_i of the set of DFA's.
  In that case the *last valid prefix* has to be returned
- What does it happen if no previous valid prefix had been found? The input w cannot be partitioned

FIB

Lexical Error Recovery

- *Panic mode*: ignoring the first character of w —and successive if necessary— until some prefix can be recognized
- Only strange characters (not in Σ) can be removed: '¿', 'ç', '@', ... in languages like C or Python
- Corrections are allowed: insert a character, replace a character by a different one, swap adjacent characters (`wihle` can be turned into `while`).
  It is a rare technique

FIB

Be Careful with the Language!

Accurately define the tokens and the syntax of a language. Some strange situations:

- in Fortran IV, the construction `DO 5 I = 1,25 ...` is the header of a loop. Changing `1,25` by `1.25` it represents an assignment to the variable `DO5I`
- if real numbers can have an empty fractional part, then the array range `10..40` will be incorrectly analyzed
- also in Fortran IV, labels are required to start at the first column $\Rightarrow$ not *free-format*
- in Python

```
while b < 10:
        print b
        a, b = b, a+b
```

FIB