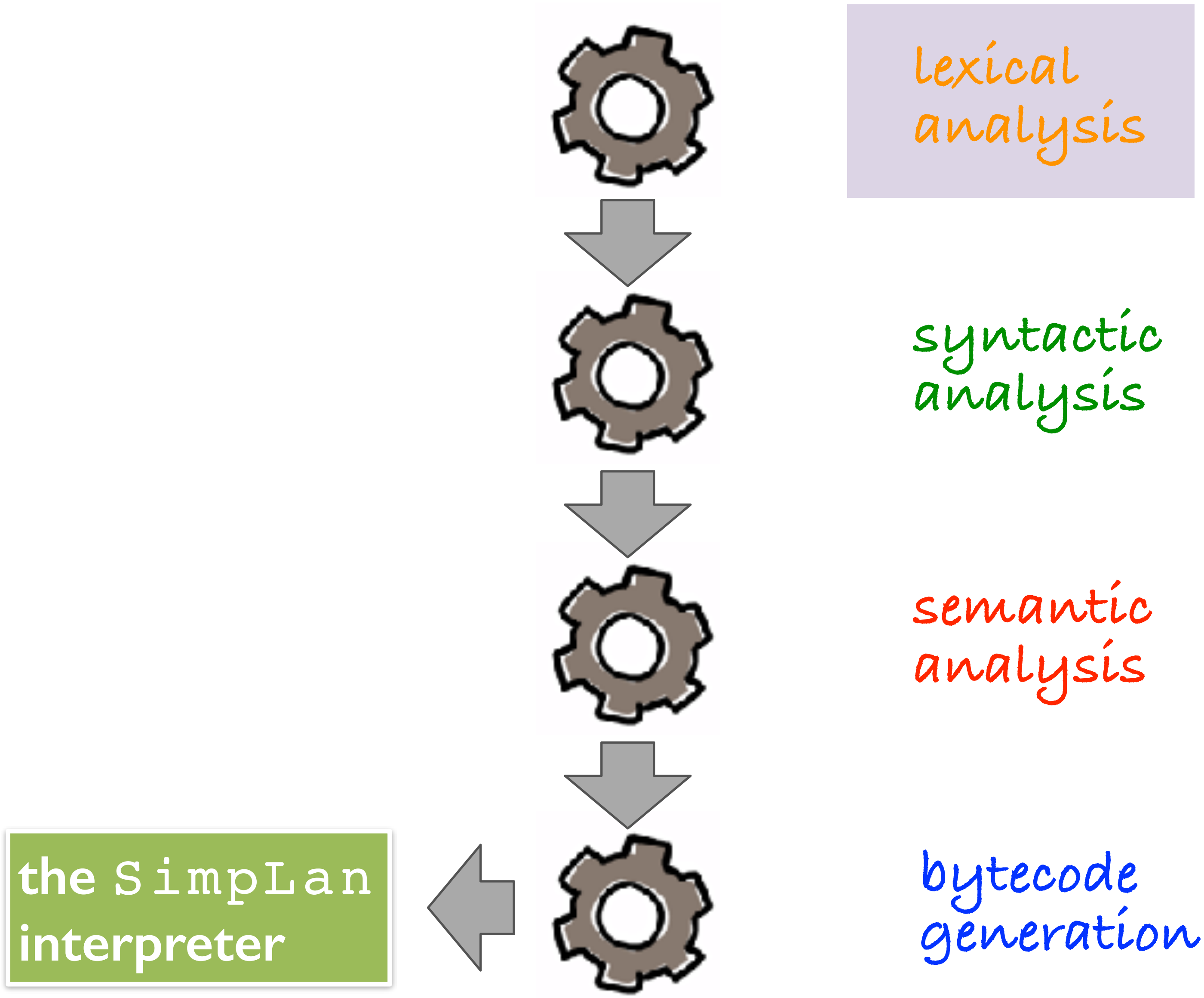
ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA
DIPARTIMENTO DI
INFORMATICA - SCIENZA E INGEGNERIA

LEXICAL ANALYSIS

COSIMO LANEVE
cosimo.laneve@unibo.it

THIS LECTURE



OUTLINE

- lexical tokens
- designing lexers by hand
- finite state automata (NFA and DFA)
- the lexer generator algorithm
- the `ANTLR` lexer

reference:

- Torben Morgensen: **Basics of Compiler Design**, chap. 2 (for the ANTLR lexer, see Terence Parr: Language Implementation Patterns)

RECOGNISING THE LEXICAL STRUCTURES

idea: breaking up very large grammars into **logical chunks**

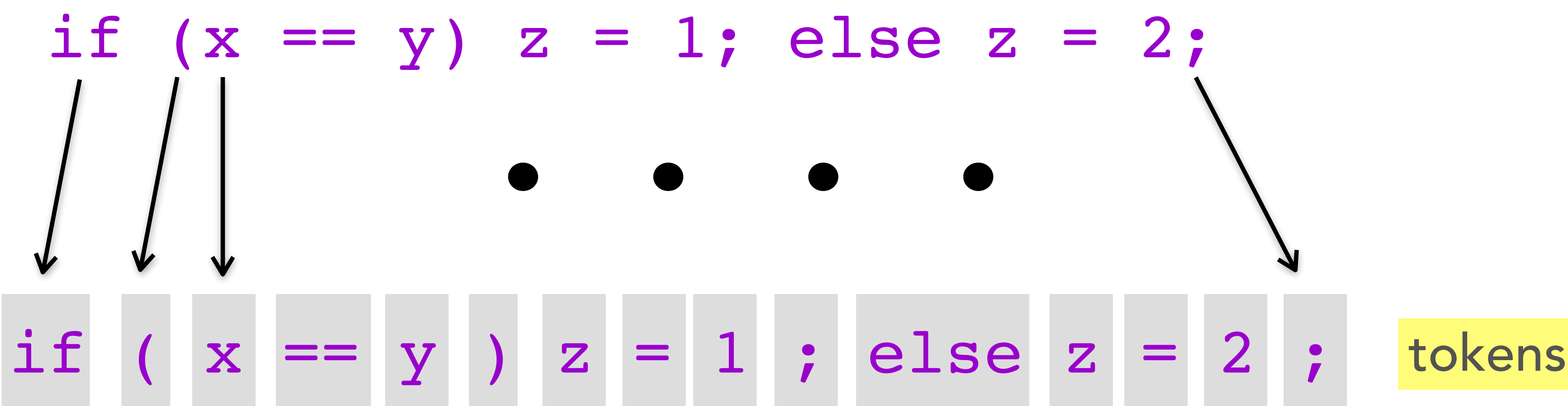
* just like we do with software

one way to do this: split a grammar into a **lexer grammar** and a **parser grammar**

* this is not a bad idea because there is a surprising amount of overlap between different languages
* **for example**, identifiers and numbers are usually the same across languages
* factoring out lexical rules into a "module" means **we can use it for different parser grammars**

LEXICAL ANALYSIS

the lexical analysis divides program texts in tokens or words



in this case the **tokens** coincide with **lexemes**

- **lexemes** also include sequences of character that are not relevant as tokens

DESIGN OF A LEXER

the **input** is just a sequence of characters

example:

```
if (x == y)
  z = 1;
else
  z = 2;
```

in this case, the input string is

```
\t if (x == y)\n\t\tz = 1;\n\telse\n\t\tz = 2;
```

goal: find the **lexemes** and map them to **tokens**:

* partition the input string into substrings (called **lexemes**), and
* classify lexemes according to their role (role = **token**)

DESIGN OF A LEXER/CONT.

the input string is

```
\t if (x == y)\n\t\tz = 1;\n\telse\n\t\tz = 2;
```

the partitioning into lexemes is

```
\t if ( x == y ) \n\t\t z = 1 ; \n\t else \n\t\t z = 2 ;
```

(**19 lexemes**! count the underlines) that are mapped to a sequence of tokens

```
IF, LPAR, ID("x"), EQUALS, ID("y"), RPAR . . .
```

remarks:

why do we need these infos?

- lexemes consisting of `\n` and `\t` are erased and do not produce tokens
- some tokens have attributes: the lexeme and/or the line number

DESIGN OF A LEXER/CONT.

it is inconvenient to built a lexer by yourself

- it is tedious repetitive, error-prone, and non-maintenable

it is much better to use a lexer generator!

- with a generator at hand, **we can focus directly to the definition of the lexemes and of the tokens**
  - that is, provide the **lexical description of the language**
- . . . and **automatically generating** the code that performs the partitioning into lexemes/tokens
  - automatically generated code may have repetitions

DESIGN OF A LEXER (BY HAND)

let's build a (simple) lexer **BY HAND** in Java

* the objective is to see **how it is done** and understanding where are the code repetitions we want to hide

our simple lexer must recognize 4 tokens

token	lexeme
ID	a sequence of one or more letters or digits starting with a letter
EQUALS	“==”
PLUS	“+”
TIMES	“*”

THE LEXER IN PSEUDOCODE JAVA

```
c = nextChar();
if (c == '=') { c=NextChar(); if (c == '=') {
                                  return EQUALS; } }
else if (c == '+') { return PLUS; }
else if (c == '*') { return TIMES; }
else if (c is a letter) {
    c = NextChar();
    while (c is a letter or digit) { c = NextChar(); }
    undoNextChar(c);
    return ID;
}
```

why do we use `undoNextChar()`?

it performs a look-ahead to determine whether the lexeme ID may be longer or not

for simplicity, we are not considering **errors**

THE MAXIMAL MATCH RULE

the previous code shows an instance of the **maximal match** rule:

- this rule is used by **every** lexer
- the rule: **the input stream of characters is partitioned into lexemes that are as longer as possible**
- example: in `Java`, "`iffy`" is not partitioned into "`if`" (the keyword `IF`) and "`fy`" (which is an `ID`), but in "`iffy`" (`ID`)

THE LEXER IN PSEUDOCODE JAVA

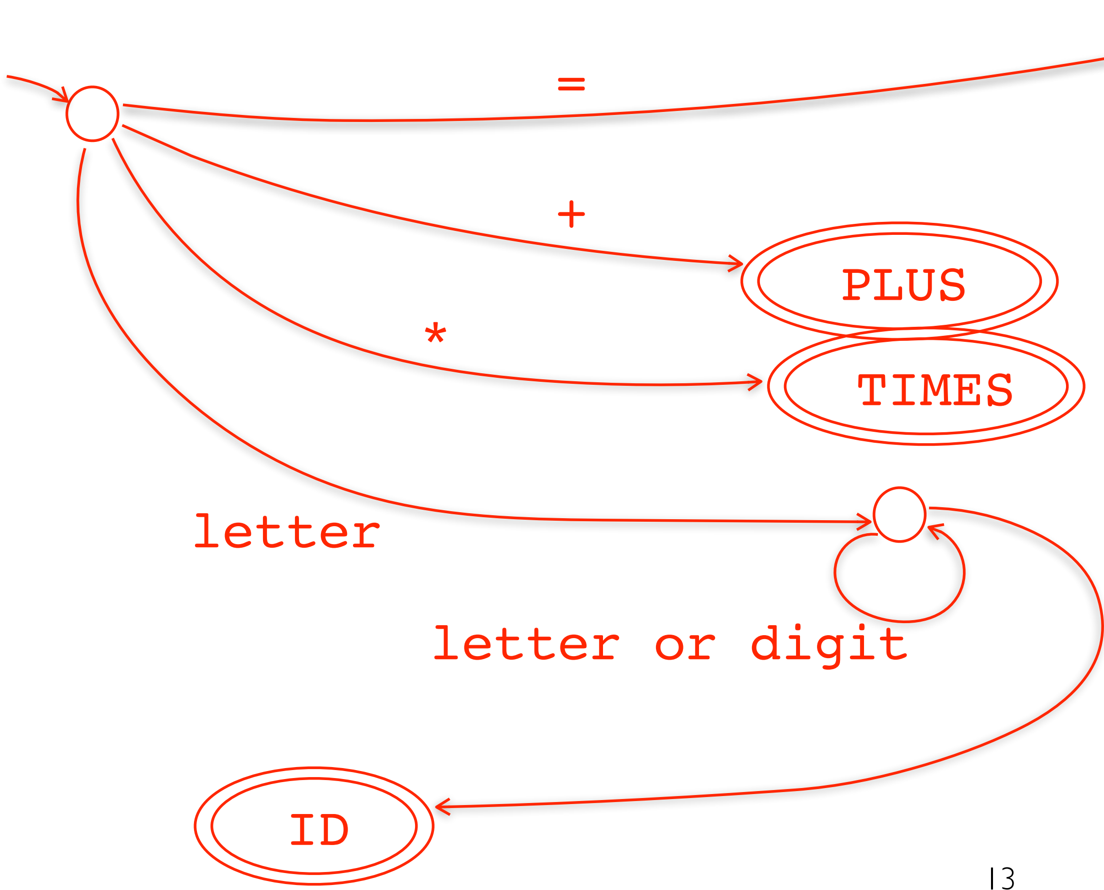
```
c = nextChar();
if (c == '=') { c=NextChar(); if (c == '=') {
                                  return EQUALS; } }
else if (c == '+') { return PLUS; }
else if (c == '*') { return TIMES; }
else if (c is a letter) {
  c = NextChar();
  while (c is a letter or digit) { c = NextChar();
   }
  undoNextChar(c);
  return ID;
}
```

the parts in red are important to **specify** the lexer

LEXER ABSTRACT MODEL

is there a computational model that allows us to define lexer's behaviour?

YES! the **nondeterministic finite state automata**



the code follows this pattern:

- read the next character and compare it with a predetermined one
- **if** there is a match **then** return a token, **otherwise** repeat until it is possible to return a token

RECAPS OF PROGRAMMING LANGUAGES: NFA

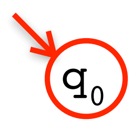
Definition: Nondeterministic Finite-state Automata

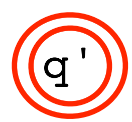
A NFA is a tuple $(Q, \Sigma, \delta, q_0, F)$ where

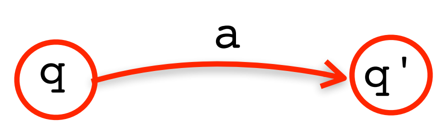
- Q is a finite set of *states*
- Σ is a finite set of symbols (the *input alphabet*)
- δ, called the *transition relation*, is a relation $Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ [instead of writing $\delta(q, a) = q'$ we write $q \xrightarrow{a} q'$]
- $q_0 \in Q$ is the *initial state*
- $F \subseteq Q$ are the *final states*

NFA have a **graphical notation**:

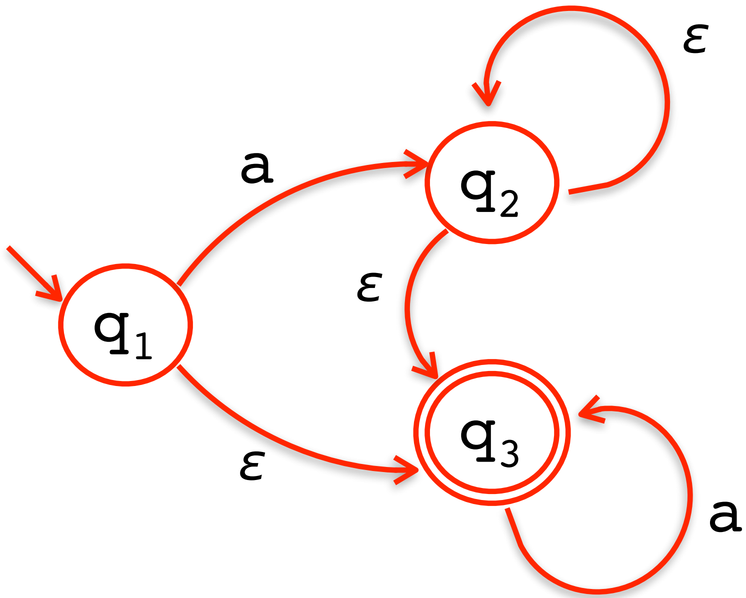
- states are denoted 

- the initial state is denoted 

- the **final states** are denoted 

- labelled transitions between **two** states 

example



as a tuple: $(\{q_1, q_2, q_3\}, \{a\}, \delta, q_1, \{q_3\})$ where

$$\delta = \{ q_1 \xrightarrow{a} q_2 \,,\; q_1 \xrightarrow{\varepsilon} q_3,\; q_2 \xrightarrow{\varepsilon} q_2,\; q_2 \xrightarrow{\varepsilon} q_3, q_3 \xrightarrow{a} q_3 \}$$

RECAPS OF PROGRAMMING LANGUAGES: NFA

Definition: language defined by an NFA

The language defined by an NFA $\mathtt{M} = (Q, \Sigma, \delta, q_0, F)$, written $\mathcal{L}(\mathtt{M})$, is the set

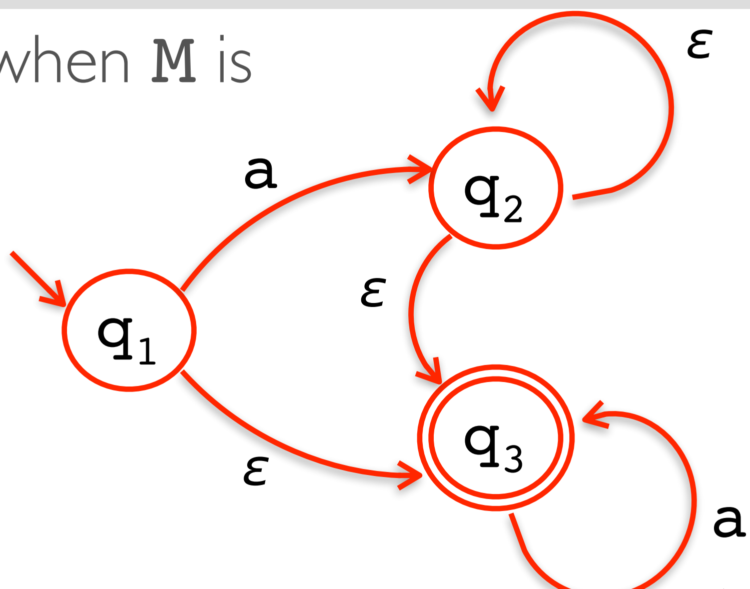
$$\{ \gamma \mid \gamma \in \Sigma^* \text{ and } [\gamma = a_1 \ldots a_n \text{ implies } (q_{i-1} \xrightarrow{a_i} q_i \in \bar{\delta})^{i \in 1..n} \text{ and } q_n \in F] \}$$

where $\bar{\delta}$ is the relation defined as follows

$$\bar{\delta}(q_1, a) = q_n \quad \text{if } q_1 \xrightarrow{\varepsilon} q_2 \xrightarrow{\varepsilon} \ldots \xrightarrow{\varepsilon} q_i \xrightarrow{a} q_{i+1} \xrightarrow{\varepsilon} \ldots \xrightarrow{\varepsilon} q_n$$

are transitions in δ

example: when $\mathtt{M}$ is



$$\mathcal{L}(\mathtt{M}) = \{\varepsilon, \mathtt{a}, \mathtt{aa}, \mathtt{aaa}, \ldots\}$$

RECAPS OF PROGRAMMING LANGUAGES: NFA

string **accepted/refused** by a NFA

* start in the unique initial state
* then start reading the input string a character at a time
* when the reading **terminates**
    * if the state where you arrive **is final** then the **string is accepted**
    * if the state where you arrive **is NOT final** then the **string is refused**
* if **no** transition is possible meanwhile, then the **string is refused**

DESIGN OF A LEXER

two parts:

PART 1: **description** (define **what** the lexer does)

- **describe every token in a precise way** — with a **formal model** such as the finite state automata
- define the **association lexeme-token** for every possible lexeme in the input language (and the corresponding **action to do**)

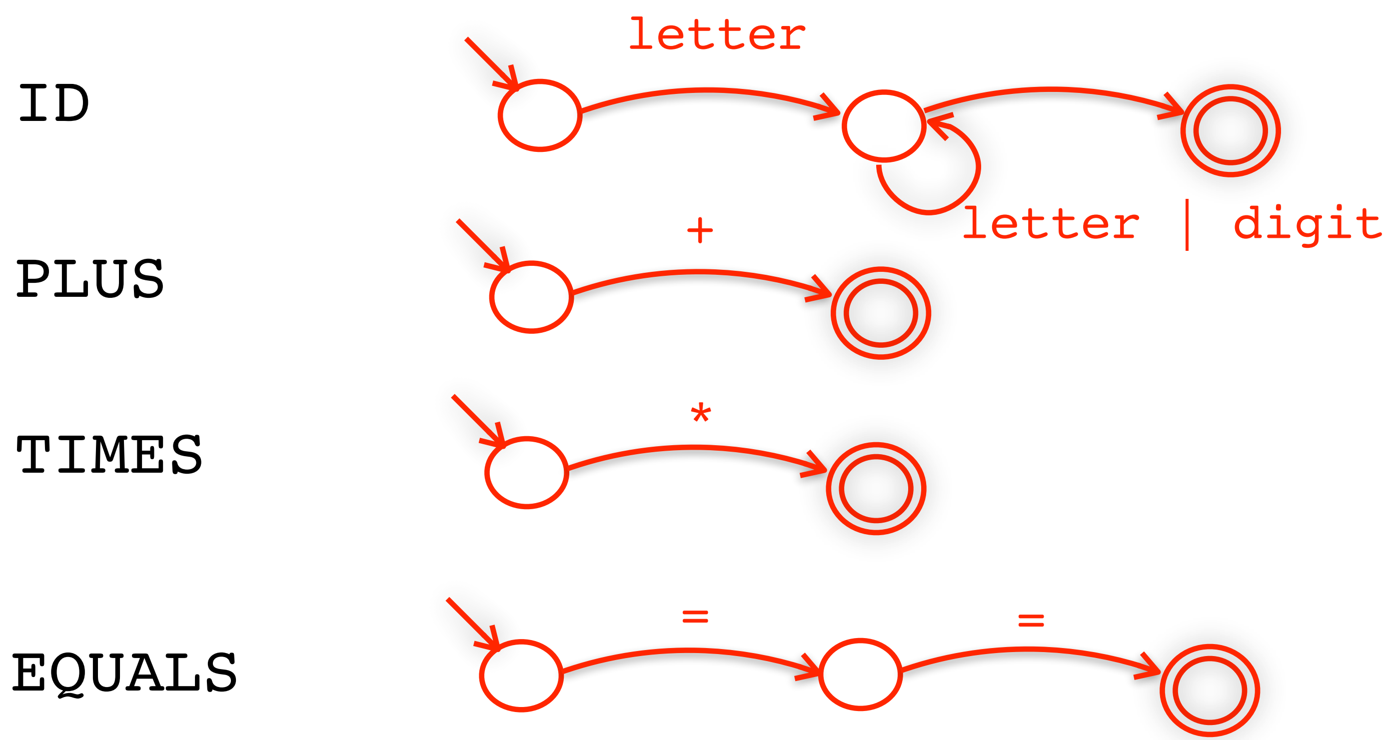
PART 2: **implementation** (define **how** the lexer behaves)

- building the automaton corresponding to the lexer — the elements that are used **are common to every lexer** (it is a library)
- define the scanner of the input program, for example `NextChar()` and `undoNextChar(c)`

PART 1: DESCRIPTION OF A LEXER

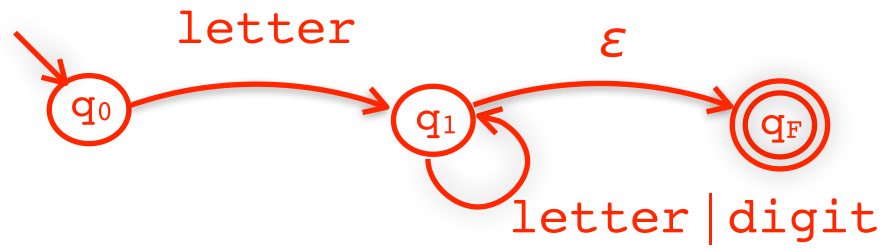
- define an **NFA** for every lexeme of the language
- associate the **NFA** to the recognized token

example:



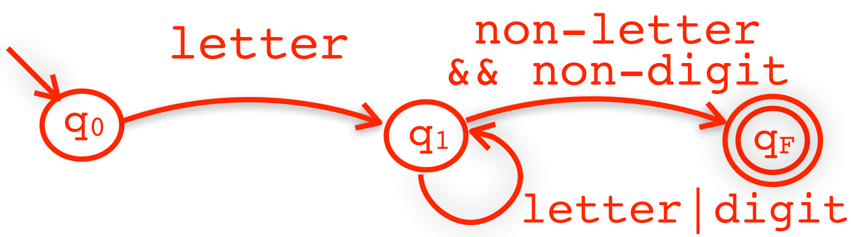
PART 2: IMPLEMENTATION OF THE LEXER

the identification of the token `ID` has an **unlabelled transition** — actually an ε-transition — the one from q_1 to q_F



* the automaton is **nondeterministic**: in the state q_1 it is not clear what happens when a letter or a digit arrives — do you transit to q_1 or you transit to q_F **without waiting for** the next character

* it is more convenient to use the **deterministic** automaton (**DFA**: Deterministic Finite-state Automata)



and use look-aheads

* for variable-length lexemes

PART 2: IMPLEMENTATION — THE ACTIONS

when a token is recognized, the NFA must execute **actions**:

- `return TOKEN` — the caller of the lexer (the parser) gets back the recognized token and the **lexer restarts from the initial state**

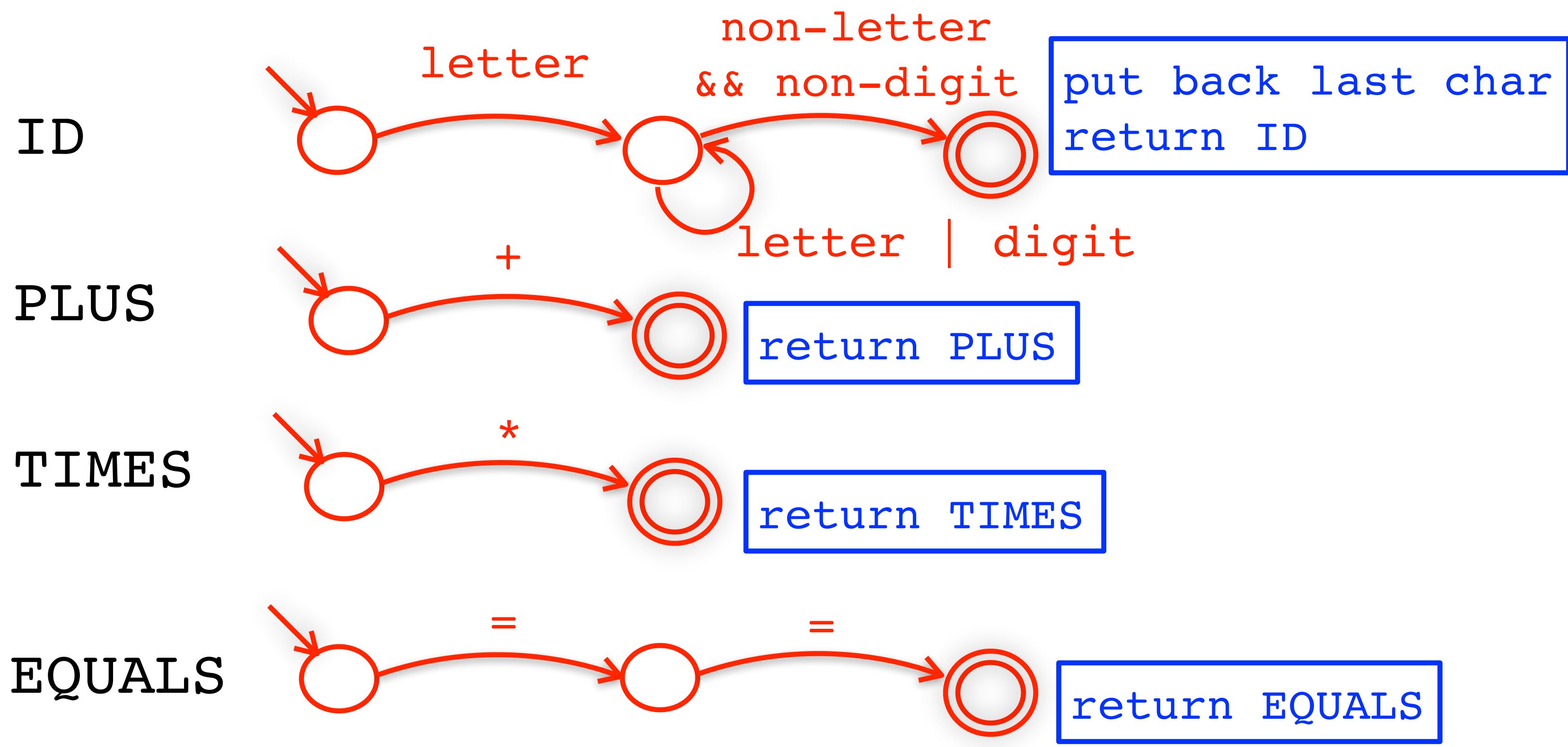
this action **resets the lexer to the initial state**
- the lexer is invoked by the parser
- every time exactly one token is returned

- **management actions of lookaheads**, for example `undoNextChar(c)` for tokens that correspond to lexemes of variable length (in our case, token `ID`)
  - see maximal match rule

PART 2: IMPLEMENTATION — THE ACTIONS

* in correspondence of the final states, we need to specify the actions of the lexer

example:

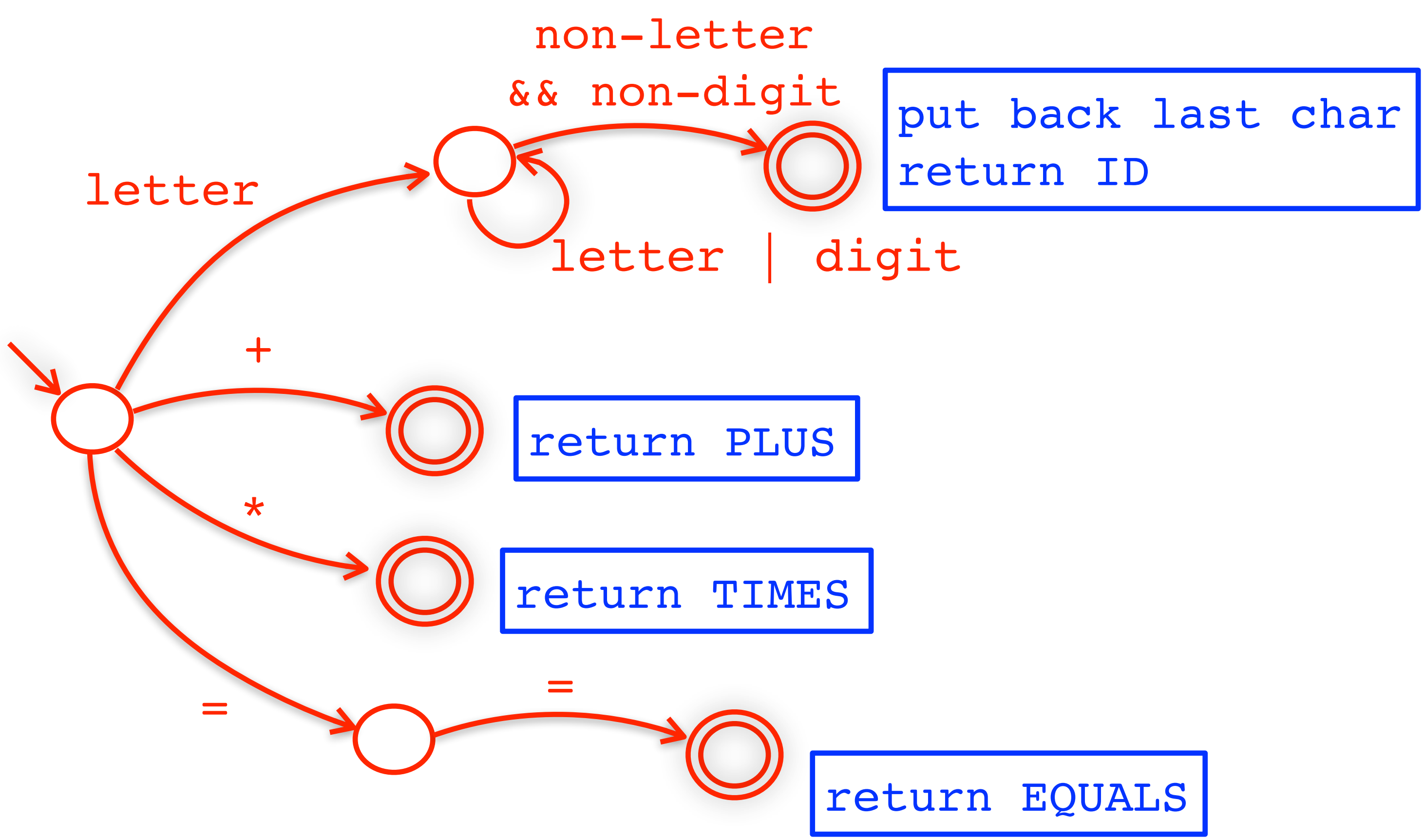


remark: the NFA of the description becomes an **extended NFA**

PART 2: COMBINE THE EXTENDED NFA

problem: the lexer must have a unique entry point

algorithm: identify the initial states of the NFA that correspond to the tokens



the combination is not exactly like this! in this case the resulting automaton is DETERMINISTIC

problem: combining the NFAs may give nondeterminism in general

PART 2: COMBINING THE EXTENDED NFA

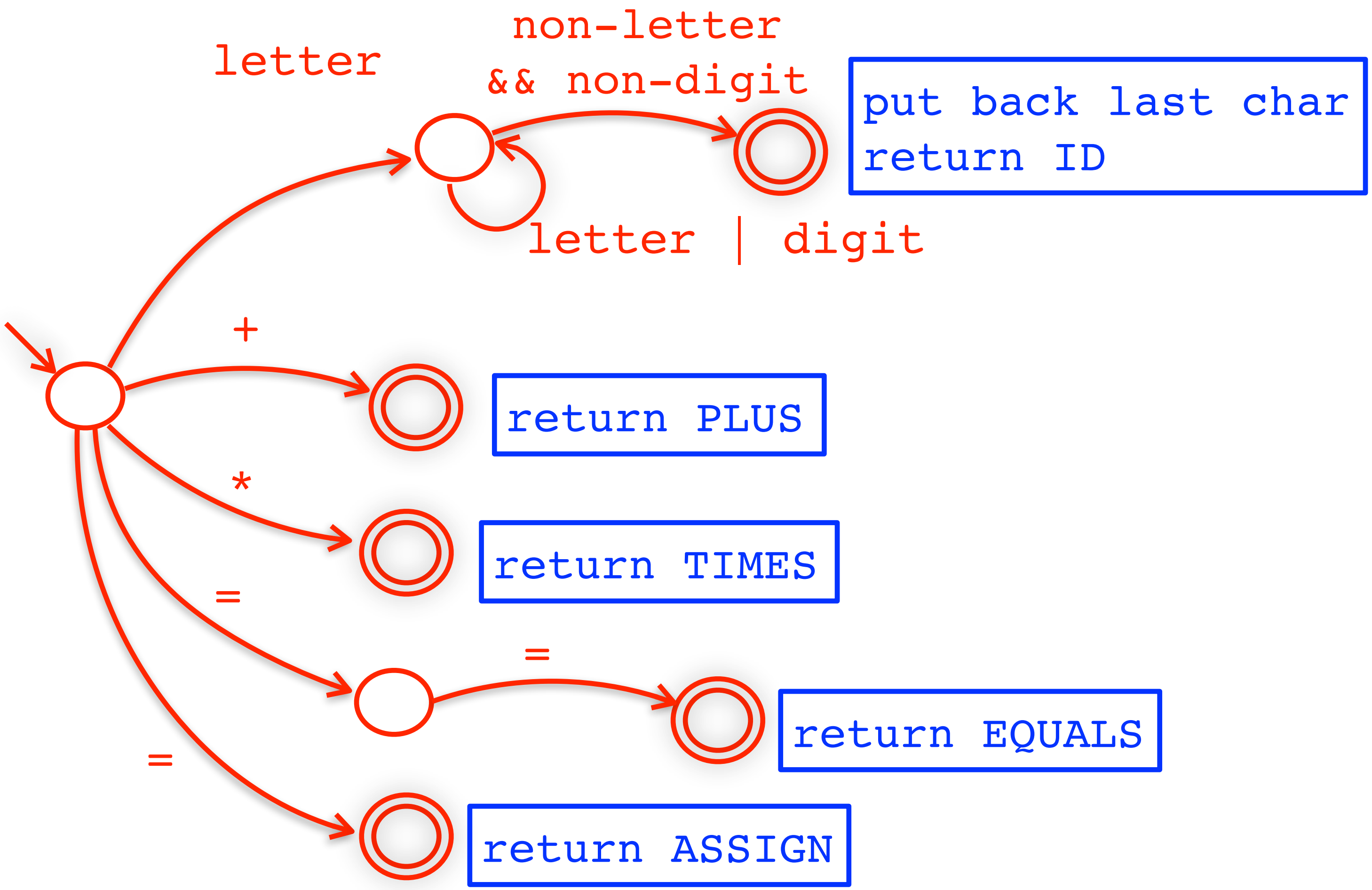
let's build a simple lexer that recognises **5 tokens**

token	lexeme
`ID`	`a sequence of one or more letters or digits starting with a letter`
`EQUALS`	`"=="`
`PLUS`	`"+"`
`TIMES`	`"*"`
`ASSIGN`	`"="`

remark: `ASSIGN` is a prefix of `EQUALS`

PART 2: COMBINING THE EXTENDED NFA

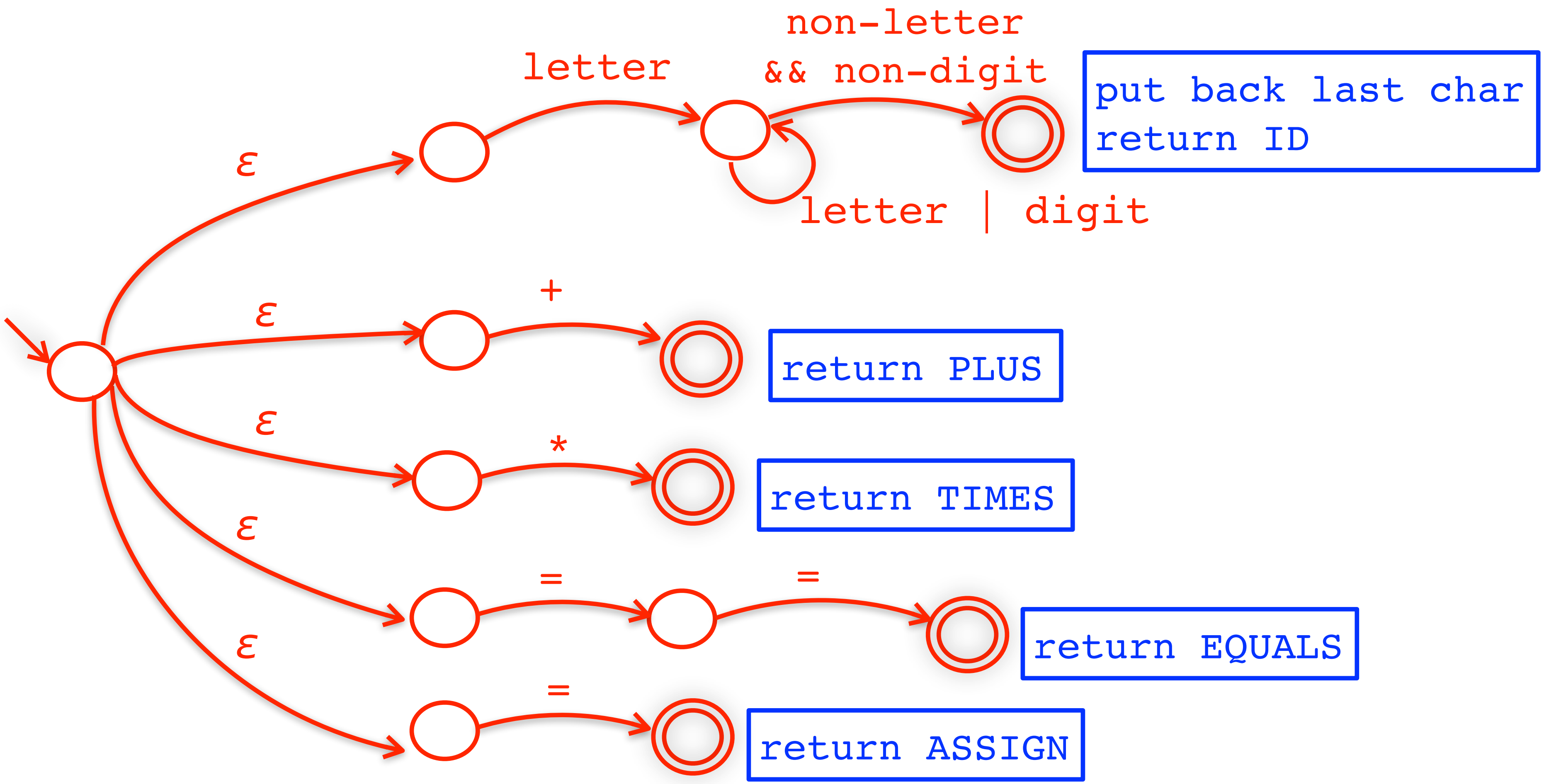
the previous algorithm gives



problem: the automaton is **NFA** — how to improve it?

PART 2: COMBINING THE EXTENDED NFA

a better solution…



LEXER'S ALGORITHM

Algorithm: lexer

1. define an NFA for every lexeme
2. combine the NFA identifying the initial states
3. if the resulting NFA in 2 is nondeterministic then **transform the automaton in deterministic (DFA)**

   + using a textual notation for DFA

4. use the following rules:
   a. when a **final state** is reached:
      i. store the position in input (therefore it is possible to read other characters)
      ii. keep reading other characters transiting from state to state

      implement them!

   b. if **other transitions are not possible** with the next character:
      i. rollback to the last final state (henceforth perform undo of the corresponding readings) and return the token corresponding to the last final state

DETERMINISTIC FINITE-STATE AUTOMATA

Definition: Deterministic Finite-state Automata

A DFA is a NFA $(Q, \Sigma, \delta, q_0, F)$ such that

- δ is a function $Q \times \Sigma \rightarrow Q$

interesing properties of DFA

Theorem: Subset Construction [Morgensen, sec 2.6]

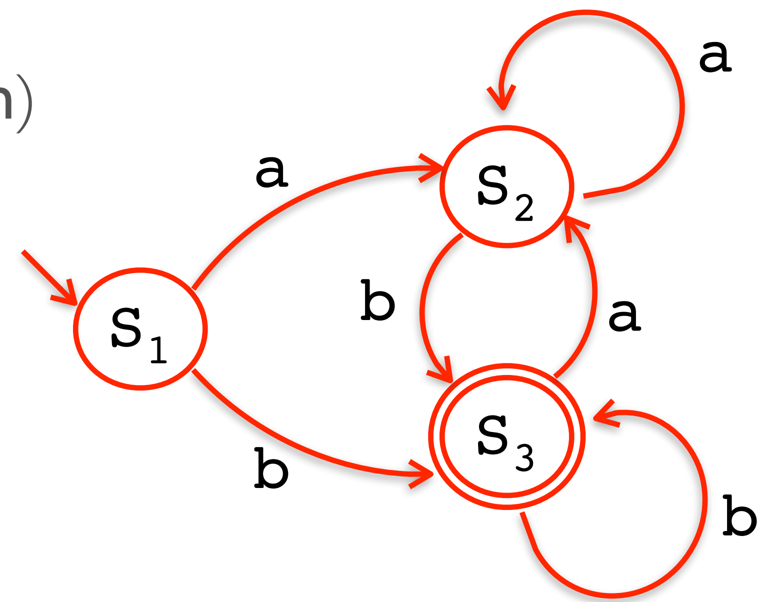
Given a NFA $\mathtt{M}$, it is possible to define a DFA $\mathtt{M'}$ such that $\mathcal{L}(\mathtt{M}) = \mathcal{L}(\mathtt{M'})$.

Theorem: Hopcroft Algorithm [Morgensen, sec 2.8]

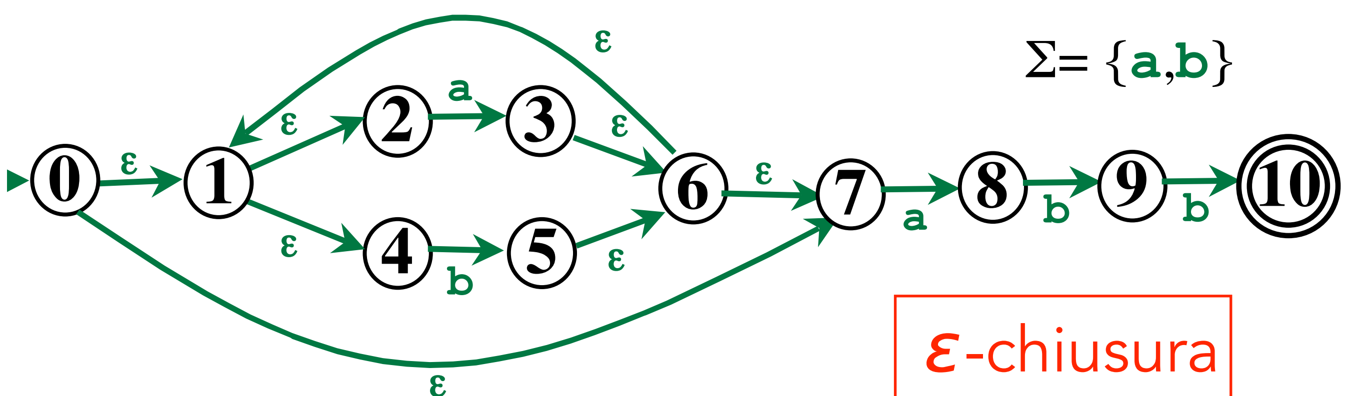
Given a DFA $\mathtt{M}$, it is possible to define a DFA $\mathtt{M'}$ with a minimal set of states such that $\mathcal{L}(\mathtt{M}) = \mathcal{L}(\mathtt{M'})$.

EXERCISES

1. minimize the DFA (cf. **Hopcroft's algorithm**)



2. transform the following NFA into a DFA (the **subset construction method**) and, in case minimize it



ε-chiusura

LEXER'S ALGORITHM — PRACTICAL REMARKS

ambiguity

- **problem**: the lexer reaches several different final states
  - example: "`if`" corresponds both to **`ID`** and to **`IF`** (reserved keyword)
- **problem**: while reading characters, the lexer automata go through several different final states

  example: "`=`" corresponds both to **`ASSIGN`** and "`==`" to **`EQUAL`**

PRINCIPLES OF LONGEST AND FIRST MATCH

Principle of longest match

A lexer always outputs the token that consumes the longest part of the input.

- this is important when reading identifiers and numbers (otherwise prefixes would be recognized as tokens, as well)

Principle of first match

Tokens are alaways prioritised, therefore the lexer can decide which token to recognize if two tokens are possible for the same input

- this is important when reading keywords (otherwise they could be recognized as identifiers)

LEXER'S ALGORITHM — PRACTICAL REMARKS

errors in the input

- **problem**: remove illegal lexemes and print an error message
  solution: ✔ **remove a character at a time** and add a lexeme that corresponds to every character
  ✔ **this lexeme has the lowest priority** — the corresponding action will be executed when no other lexeme is recognized

remove white spaces `\n` , `\t` **e** `\r`

- **solution**: the final states of lexemes with these characters are **special** — they do not return a token but recursively invoke the lexer (= going back to the initial state)

remark about the lookaheads

- lookaheads may have **whatever length**; in case you need to perform `undoNextChar(c)` several times

LEXER'S ALGORITHM

the one at pag 25 is the algorithm used by **every lexer**

* the **description** is the step 1 — this is **the part which is required to the language designer**!
* the **implementation** is the steps 2, 3 and 4 — this is the **part that is performed automatically** by a lexer generator

how to specify the automata at step 1?

HOW TO SPECIFY THE AUTOMATA: THE REGULAR EXPRESSIONS

the automata allow us to define the lexemes that correspond to a token **in a visual manner**

- but **they are not adequate** as specification language

an **equivalent description** to the automata (DFA and NFA) are the **regular grammars/regular expressions**

- regular grammars/expressions are a compact way for defining a language that is accepted by a FA

the regular grammars/expressions **are used as input** to the lexer generators

- define every lexeme, including white space sequences and comments, which must be recognized but **not** associated to a token, in such a way to be able to ignore them

REGULAR GRAMMARS

Definition: regular grammar

A grammar (**N**, **T**, →, S) is **regular** if its productions → have the form

- A → a
- A → aB
- A → ε

- in the **literature**, regular grammars are also called right-linear grammars
- **example**: Java identifier definition as regular grammar

```
ID → ('a'..'z'| 'A'..'Z') CONT
CONT → ('a'..'z'| 'A'..'Z'| '0'..'9' | '_') CONT
CONT → ε
```

```
CONT = (LETTER | DIGIT | '_')*
```

REGULAR GRAMMARS AND DFA

THEOREM: from DFA to regular grammars

for every finite automata $\mathtt{M}$, there is one regular grammar $\mathcal{G}$ where $\mathscr{L}(\mathtt{M}) = \mathscr{L}(\mathcal{G})$

Algorithm: from DFA to regular grammar

the nonterminals of the grammar are the states of the automata (written in capital letters, for simplicity)

the productions are

- if $q \xrightarrow{a} q'$ in the automata and q' is not final then Q ⟶ a Q' in the grammar
- if $q \xrightarrow{a} q'$ in the automata and q' is final then Q ⟶ a Q' | a in the grammar
- if q is initial and final then Q ⟶ ε in the grammar

EXAMPLE: A JAVA IDENTIFIER AS A REGULAR EXPRESSION

lexical definition (in **English**):

- a letter, followed by zero o more letters, digits or symbols '_'

lexical definition (as **regular expression**):

```
LETTER (LETTER | DIGIT | '_')*
```

ε	means "empty string"
|	means "or"
string1 string2	means "sequence"
*	means "repeat 0 or more times"
( )	means "grouping"

remark: there is a precedence among the regular expressions operators:

* has precedence on concatenation that has precedence on |

LANGUAGE DEFINED BY A REGULAR EXPRESSION

the **language defined by a regular expression** is the set of strings that **match with** the expression

examples

regular expressions	corresponding language
`'00' | '1' | ε`	`{00, 1, ε}`
`'0'*`	`{ε, 0, 00, 000, …}`
`ε*`	`{ε}`
`('0'|'1')*`	`{ε, 0, 1, 00, 01, 10, …}`
`('0'|'1')('0'|'1')*`	`{0, 1, 00, 01, 10, …}`
`('1'|ε)('01')*('0'|ε)`	`sequenze anche vuote di 0 e 1 alternati`

OPERANDS OF A REGULAR EXPRESSION

the operands

- correspond to the labels of the transitions of the FA
- are single characters between apices or sequences of characters between apices, examples: `'a'` and `'while'`
- are the special character ε (the empty string)

example:

```
letter: 'a'| 'b' | 'c' | ... | 'z' | 'A' | ... | 'Z'
digit: '0' | '1' | ... | '9'
```

in many lexers (included ANTLR) you can also write

```
letter: ('a'..'z') | ('A'..'Z')
digit: ('0'..'9')
```

OTHER USEFUL OPERATORS OF REGULAR EXPRESSIONS

- operator `+` (one or more repetitions)

  ```
  natural_numbers: digit+ ;
  ```

  note: `digit+` is equal to `digit (digit)*`

- operator `?` (zero or one repetition)

  ```
  integer: ('+' | '-')? natural_numbers
  ```

  note: `('+' | '-')?` is equal to ε `|'+' | '-'`

- (`ANTLR`) operator `~('a'..'z')` are the characters that are different from `'a'..'z'`

- (`ANTLR`) operator `.` stands for every character (therefore `.*` is every sequence of characters)

LEXER GENERATORS

input: the regular expressions describing the lexemes

generate code (`C`, `C++`, `Java`, ...) that implements the full lexer algorithm:

* translate the regular expressions into FA
* merge the FA into a unique automaton
* translate the merged automaton to a Deterministic FA (more efficient to be simulated)
* produce the code that implements the "special" simulation of the DFA (lookahead for maximal match rule, priorities in case of multiple match, operations to be executed upon matching, and return to initial state)

LEXER IMPLEMENTATION

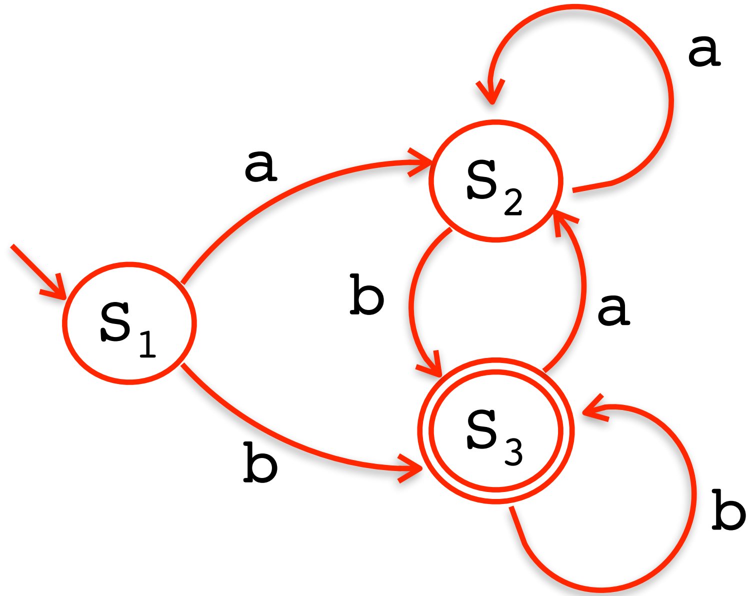
a DFA can be implemented by a 2-dimension table — let it be `T`

- a dimension describes the "automata states"
- the other dimension describes the "input symbols"
- for every transition of the automata $S_i \xrightarrow{a} S_k$, it is sufficient to define `T[i,a] = k`

the execution of the DFA **is very efficient**

- if the automata is in the state S_i and the input character is `'a'`, then read `T[i,a] = k` and jump to state S_k

EXAMPLE OF TABLE IMPLEMENTING A DFA



	a	b
s_1	s_2	s_3
s_2	s_2	s_3
s_3	s_2	s_3

`ANTLR` LEXER

the **`ANTLR`** lexer (as every lexer)

* reads the characters until one rule is selected
* then print the corresponding token
* and then restart from the next character

few relevant things (see next slides)

* the rule used is the **first longest match**
* **the lexer does not backtrack** — it never changes the previous decisions

ANTLR LEXER: THE FIRST LONGEST MATCHING RULE

if there are several rules that match with the the input, the one which is selected is that corresponding to the longest string

example:

```
SHORTTOKEN: 'abc';
LONGTOKEN: 'abcabc';
```

in **ANTLR**, the non terminals that start with an uppercase letter are the lexical (token) rules

both **SHORTTOKEN** and **LONGTOKEN** match with the initial part of the string **abcabc**

- since **LONGTOKEN** has 6 characters and **SHORTTOKEN** has only 3, the lexer returns **LONGTOKEN**
- if there are more than one rule that match, the returned one is the first in the list

example:

```
SHORTTOKEN: 'a';
FIRSTTOKEN: 'abc';
SAMELENGTHTOKEN: 'ab'. ;
```

with input **abc**, **ANTLR** selects **FIRSTTOKEN**

`ANTLR` LEXER: IRREVERSIBLE DECISIONS

once the decision is taken, the lexer **does not** rollback

example: in **theory** the grammar

```
SHORT: 'aaa';
LONG: 'aaaa';
```

might split the input `aaaaaa` in the sequence `SHORT SHORT`

but

* the lexer will choose the longest match and therefore recognise `LONG`
* since the tailing `aa` does not match with any token, the lexer will output the errors:

```
start:1:4: token recognition error at: 'aa\n'
```

ANTLR LEXER: USUAL ERRORS

the lexer chooses the next token by consuming the characters in input **without matching completely the input**, by erasing the shortest rules

> if the selected token does not match with the input, then an error is reported

example: the input `abcabQ` with the grammar

```
SHORTTOKEN: 'abc';
LONGTOKEN: 'abcabc';
```

* `SHORTTOKEN` matches with 3 characters, `LONGTOKEN` matches with more than 4 characters
* henceforth `ANTLR` chooses `LONGTOKEN`
* unfortunately `LONGTOKEN` does not match with the input and therefore the lexer backtracks and recognizes `SHORTTOKEN` giving an error for `abQ`

ANTLR LEXER: USE OF PUSHDOWN AUTOMATA

ANTLR lexer uses the same technique for the lexer and the parser

- therefore you may write lexical clauses using LL(*) grammars
- the lexer becomes less efficient

example: the lexical part of the grammar

```
init : TOKEN (',' TOKEN)* ;

TOKEN : 'a'TOKEN'b' | 'a''b' ;
WS : (' ' | '\n' | '\r' | '\t')+ -> skip ;
```

DON'T DO IT!

is correct!

these are called **SCANNERLESS PARSERS**

ANTLR — AN EXAMPLE

rules defined with `fragment` do not generate nodes in the syntax tree (no token is generated!)

- CHAR and DIGIT must be invoked by other lexer rules
- se non si mette `fragment`, 'a' è riconosciuto come CHAR

```
grammar Example;
// THIS IS THE INPUT FOR THE PARSER

. . .

// THIS IS THE INPUT FOR THE LEXER
fragment CHAR     : 'a'..'z' |'A'..'Z' ;
ID                : CHAR (CHAR | DIGIT)* ;
fragment DIGIT    : '0'..'9';
NUMBER            : DIGIT+;
// ESCAPE SEQUENCES
WS              : (' '|'\t'|'\n'|'\r')-> skip;
LINECOMENTS     : '//' (~('\n'|'\r'))* -> skip;
BLOCKCOMENTS    : '/*'( ~('/'|'*')|'/'~'*'|'*'~'/'|BLOCKCOMENTS)* '*/' -> skip;
ERR             : .  -> channel(HIDDEN) ;
```

the lexer rhs are regular expressions!

no node in the syntax tree is generated! The characters are skipped

ANTLR ha diversi canali di output, quello standard è 0 e i simboli/token si possono recuperare dai diversi canali

NEXT LECTURE