UNIVERSITÀ di VERONA
Dipartimento di INFORMATICA

Lexical Analysis: Regular Expressions & Finite Automata

Alessandra Di Pierro
alessandra.dipierro@univr.it
Using Cosmin E. Oancea and Jost Berthold's material (DIKU - University of Copenhagen)

Department of Computer Science
University of Verona

Compilers Lecture Notes

Structure of a Compiler

Program text

↓

Lexical analysis

↓

Symbol sequence

↓

Syntax analysis

↓

Syntax tree → Intepretation

↓

Typecheck

↓

Syntax tree

↓

Intermediate code generation ⟶ Intermediate code

↑

Machine code generation

↑

Symbolic machine code

↑

Register allocation

↑

Ditto with named registers

↑

Assembly and linking

↑

Binary machine code

Lexical Analysis

Lexical: relates to the words of the vocabulary of a language, (as opposed to grammar, i.e., correct construction of sentences).

- "My mother coooookes dinner not."
- Lexical Analyzer, a.k.a. lexer, scanner or tokenizer, splits the input program, seen as a stream of characters, into a sequence of tokens.
- Tokens are the words of the (programming) language, e.g., keywords, numbers, comments, parenthesis, semicolon.
- Tokens are classes of concrete input (called lexeme).

Constructing a Lexical Analyser

- By hand: Identify *lexemes* in input and return *tokens*
- Automatically: Lexical-Analyser generator: it compiles the patterns that specify the lexemes into a code (the lexical analyser).
- First we need to introduce:
  - Regular expressions
  - Non-deterministic automata
  - Deterministic automata

Scanning and Parsing

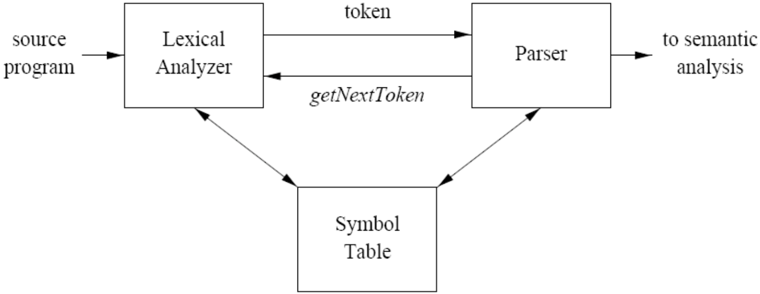


Figure 3.1: Interactions between the lexical analyzer and the parser

Important Notions

- Token: pair consisting of (`token-name`, `opt-value`)
- Pattern: form of the lexemes for a token
- Lexemes: sequence of characters matching the pattern for a token

Example

```
printf("Total = %d/n", score);
```

`printf` and `score` are lexemes for token **id** that matches pattern in Table 3.2

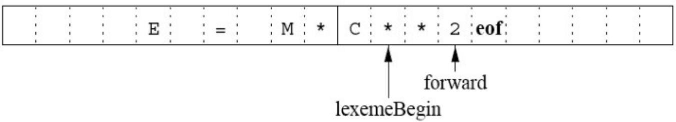
Classes of tokens

Token	Informal Description	Sample Lexemes
if	characters `i`, `f`	`if`
else	characters `e`, `l`, `s`, `e`	`else`
comparison	`<` or `>` or `<=` or `>=` or `==` or `!=`	`<=`, `!=`
id	letter followed by letters and digits	`pi`, `score`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything but `"`, surrounded by `"`'s	`"core dumped"`

Figure 3.2: Examples of tokens

Reading the Input

- Reading a large source program one character by one is very slow.
- Specialised buffering techniques have been developed to speed up the process.
- Using a buffer of size N (e.g. $N = 4096$ bytes, the typical size of a disk block), we can read N character instead of one.
- Using a two buffers scheme we can handle large lookaheads safely.



Using a pair of input buffers

Formalism

Definition (Formal Languages)

Let $\sum$ be an alphabet, i.e., a finite set of allowed characters.

- **A word** *over $\sum$ is a string of chars $w = a_1 a_2 \ldots a_n$, $a_i \in \sum$*
  $n = 0$ is allowed and results in the empty string, denoted ϵ.
  $\sum^*$ is the set of all words over $\sum$.
- **A language** *L over $\sum$ is a set of words over $\sum$, i.e., $L \subset \sum^*$.*

Examples over the alphabet of small latin letters:

- $\sum^*$ and $\emptyset$
- All C keywords: $\{$`if, else, return, do, while, for, ...`$\}$
- $\{a^n b^m\}$, $\forall n, m \geq 1$, i.e., $\{a \ldots ab \ldots b\}$
- $\{a^n b^n\}$, $\forall n \geq 0$
- All palyndromes: $\{$`kayak, racecar, mellem, retter`$\}$
- $\{a^n b^n c^n\}$, $\forall n \geq 0$

Languages

Aim of compiler's front end: decide whether a program respects the language rules.

Lexical analysis: decides whether the individual tokens are well formed, i.e., requires the implementation of a simple language.

Syntactical Analysis: decides whether the composition of tokens is well formed, i.e., more complex language that checks compliance to grammar rules.

Type Checker: verifies that the program complies with (some of) the language semantics.

Language Examples: Number Literals in C++

- Integers in decimal format: `234`, `0`, `8` but not `08` or `abc`!
- Integers in hexadecimal format: `0X123`, `0xcafe` but not `0X`, `0XG`!
- Floating point decimals: `0.` or `.345` or `123.45`.
- Scientific notation: `234E-45` or `0.E123` or `.234e+45`.

- A decimal integer is either `0` or a sequence of digits (`0-9`) that does not start with `0`.
- A hexadecimal integer starts with `0x` or `0X` and is followed by one or more hexadecimal digits (`0-9` or `a-f` or `A-F`).
- Floating-point constants have a "mantissa" and an "exponent". The mantissa is a sequence of digits followed by a period, followed by an optional sequence of digits. The exponent, if present, is specified using e or E followed by an optional sign (+ or -) and a sequence of digits. If an exponent is present, decimal point is unnecessary in whole numbers.

http://msdn.microsoft.com/en-us/library/tfh6f0w2.aspx.

Regular Expressions

We need a formal, compositional (& intuitive) description of what tokens are, AND automatic implementation of the token language.

Definition (Regular Expressions)

The set $RE(\sum)$ of regular expressions over alphabet $\sum$ is defined:

- *Base Rules (Non Recursive):*
  - $\epsilon \in RE(\sum)$ *describes the lang consisting of only the empty string.*
  - $\mathtt{a} \in RE(\sum)$ *for* $\mathtt{a} \in \sum$ *describes the lang. of one-letter word* $\mathtt{a}$.
- *Recursive Rules: for every* $\alpha, \beta \in RE(\sum)$
  - $\alpha \cdot \beta \in RE(\sum)$, *two language sequence/concatenation in which the first word is described by* α, *the second word by* β.
  - $\alpha \mid \beta \in RE(\sum)$, *alternative/union: lang described by* α *OR* β.
  - $\alpha^* \in RE(\sum)$, *repetition: zero or more words described by* α.

- One may use parenthesis `(...)` for grouping regular expressions.
- Sequence binds tighter than alternative: $\mathtt{a|bc^*} = \mathtt{a|(b(c^*))}$.

Demonstrating Regular-Expression Combinators

$\alpha \cdot \beta$ Assume the language of regular expression α and β are `L(`α`)={"a","b"}` and `L(`β`)={"c","d"}`, respectively.

Then `L(`$\alpha \cdot \beta$`)={"ac", "ad", "bc", "bd"}`.

K-word '`if`' is the concatenation of two regular expressions: '`i`' and '`f`'.

α^* Assume the language of regular expression α is `L(`α`)={"a","b"}`.

Then

`L(`α^*`)={"","a","b","aa","ab","ba","bb","aaa",...}`.

Examples: Integers and Variable Names in C++

- Integers in decimal format: `234`, `0`, `8` but not `08` or `abc`!
- Integers in hexadecimal format: `0X123`, `0xcafe` but not `0X`, `0XG`!
- A variable name consists of letters, digits and underscore, and it must begin with a letter or underscore.

- Integers in decimal format:
  `(1|2|...|9)(0|1|2|...|9)* | 0`
  Shorthand via character range (`[-]`): `[1-9][0-9]* | 0`
- Integers in hexadecimal format:
  `0 (x|X) [0-9a-fA-F][0-9a-fA-F]*`
  Shorthand via at least one (+): `0 (x|X) [0-9a-fA-F]+`.
- Variable names: `[a-zA-Z_][a-zA-Z_0-9]*`

Useful Abbreviations for Regular Expressions

- Character Sets: $[\mathtt{a}_1\mathtt{a}_2\ldots\mathtt{a}_n]$:= $(\mathtt{a}_1 \mid \mathtt{a}_2 \mid \ldots \mid \mathtt{a}_n)$, i.e., one of $\mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_n \in \sum$.
- Negation: $[\hat{}\ \mathtt{a}_1\mathtt{a}_2\ldots]$ describes any $\mathtt{a} \in \sum \setminus \{\mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_n\}$.
- Character Ranges: $[\mathtt{a}_1\text{-}\mathtt{a}_n]$:= $(\mathtt{a}_1 \mid \mathtt{a}_2 \mid \mathtt{a}_n)$, where $\{\mathtt{a}_i\}$ is ordered, i.e., one character in the range between $\mathtt{a}_1$ and $\mathtt{a}_n$.
- Optional Parts: $\alpha?$:= $(\alpha|\epsilon)$ `for` $\alpha \in RE(\sum)$, optionally a string described by α.
- Repeated Parts: α^+ := $(\alpha\alpha^*)$ `for` $\alpha \in RE(\sum)$, at least ONE string describing α (but possibly more).

Some Simple Examples

- Using the alphabet of decimal digits, give regular expressions describing the following languages:
  - Numbers divisible by 5,
  - Numbers in which digit '5' occurs exactly three times.
- Are the following languages over the alphabet of decimal digits regular?
  - Numbers of arbitrary length which contain digit '1'exactly as many times as digit '2',
  - Numbers $N < 1.000.000$ which contain digit '1' exactly as often as digit '2'.

Example: A Grammar for branching statements

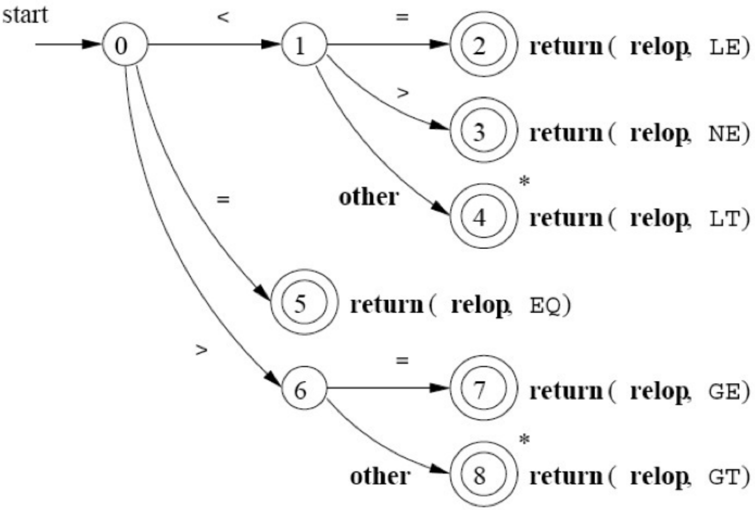
$$
\begin{array}{rcl}
\mathit{stmt} & \rightarrow & \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{stmt} \\
& | & \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{stmt}\ \textbf{else}\ \mathit{stmt} \\
& | & \epsilon \\
\mathit{expr} & \rightarrow & \mathit{term}\ \textbf{relop}\ \mathit{term} \\
& | & \mathit{term} \\
\mathit{term} & \rightarrow & \textbf{id} \\
& | & \textbf{number}
\end{array}
$$

Recognition of Tokens

$$
\begin{array}{rcl}
\mathit{digit} & \rightarrow & \texttt{[0-9]} \\
\mathit{digits} & \rightarrow & \mathit{digit}^{+} \\
\mathit{number} & \rightarrow & \mathit{digits}\ (\texttt{.}\ \mathit{digits})?\ (\ \texttt{E}\ \texttt{[+-]}?\ \mathit{digits}\ )? \\
\mathit{letter} & \rightarrow & \texttt{[A-Za-z]} \\
\mathit{id} & \rightarrow & \mathit{letter}\ (\ \mathit{letter} \mid \mathit{digit}\ )^{*} \\
\mathit{if} & \rightarrow & \texttt{if} \\
\mathit{then} & \rightarrow & \texttt{then} \\
\mathit{else} & \rightarrow & \texttt{else} \\
\mathit{relop} & \rightarrow & \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{=} \mid \texttt{<>}
\end{array}
$$

Figure 3.11: Patterns for tokens of Example 3.8

Transition diagrams



Transition diagram for **relop**

Nondeterministic Finite Automata (NFA)

NFA: a stepping stone in implementing regular expressions,

- nondeterministic: still not quite close to "real machine",
- but can be transformed to deterministic FA, which efficiently execute on real hardware,

NFA: a machine with a finite number of

- states: intial, final, intermediate,
- transitions between states: labelled with a char $c \in \sum$, or with ϵ,

NFA: used to decide if the input string is a member of the language L:

initial state where execution starts, a.k.a. starting state,

final states where execution ends if the input $\in L$, (accepting states),

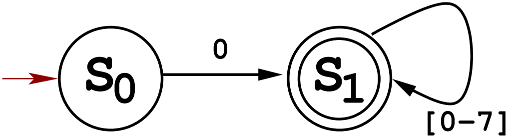
transitions: used to reach a new state from the current state,

- either follow an ϵ transition (no char of input is consumed) or
- consume the current char c of the input and follow a transition labelled with c from the current state (if such transition exists),

membership if there is a possibility to reach an accepting state after all input characters were consumed then input $\in L$.
Possibility refers to choices on what transition to follow!

NFA for Octal Number

Regular Expression for an Octal Number: `0 [0-7]*`



- Start in state S_0
    - `IF` input is 0 go to state S_1
    - `OTHERWISE` analysis fails!
- In state S_1:
    - `IF` input in $0 \ldots 7$ stay there
    - `OTHERWISE` analysis fails!
    - If end of input reached then success: an octal number was identified!

NFA Definition & Representation

Definition (NFA)

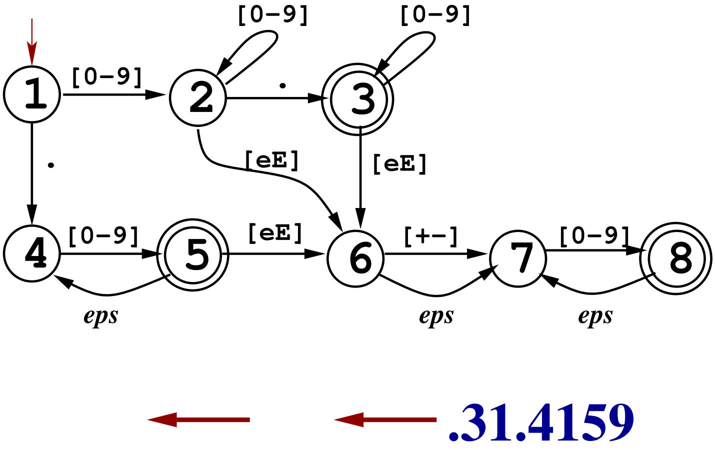
Let $\sum$ be an alphabet of (input) characters. An NFA consists of

- *An alphabet $\sum$ of input characters,*
- *A finite set S of states:*
- *A start (initial) state $s_0 \in S$ (in rep: pointed by a clean arrow)*
- *A set of final (accepting) states $F \subseteq S$ (in rep: double circles)and*
- *A relation $T \subseteq S \times (\sum \cup \{\epsilon\}) \times S$ describing state transitions.*

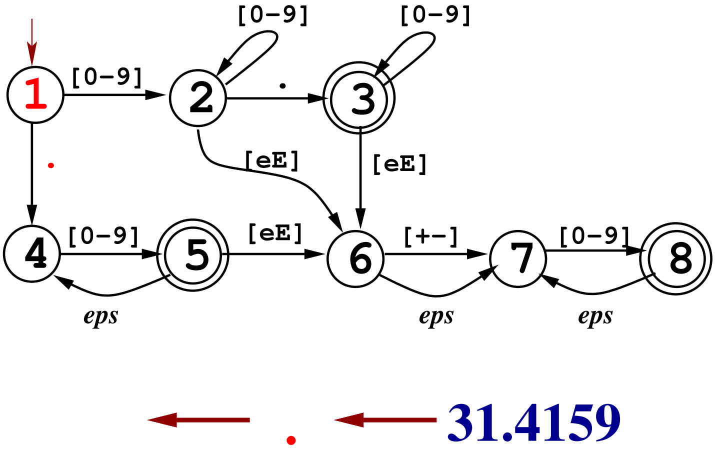
Transitions: an arrow labelled with either a char $c \in \sum$ or ϵ.Notation

- $s_i^c s_j \in T$: fire transition from state s_i to s_j on char c.
- $s_i^\epsilon s_j \in T$: if in s_i you may freely transition to s_j (without consuming an input char).
- Several options may exist, hence T is a relation.
- Acceptance: if there exist a sequence of choices reaching an accepting state at the end of the input string.

A More Complex Analysis: Float Numbers

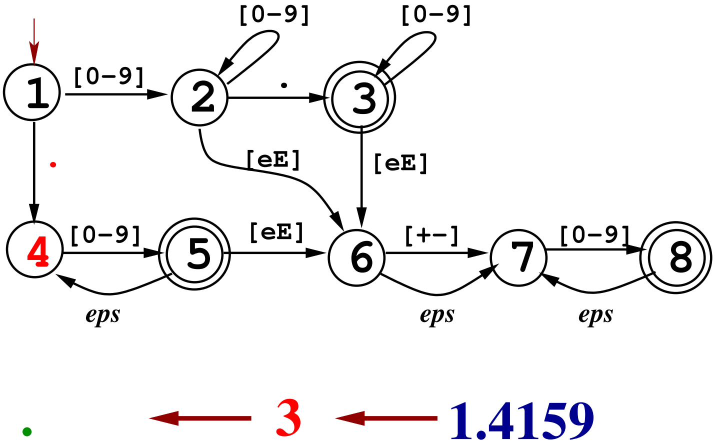


A More Complex Analysis: Float Numbers



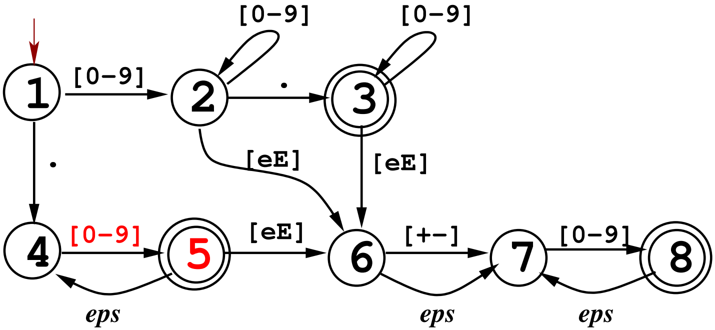
Starting at the pointed state,
transitions to new state possibly reading input.

A More Complex Analysis: Float Numbers



Starting at the pointed state,
transitions to new state possibly reading input.

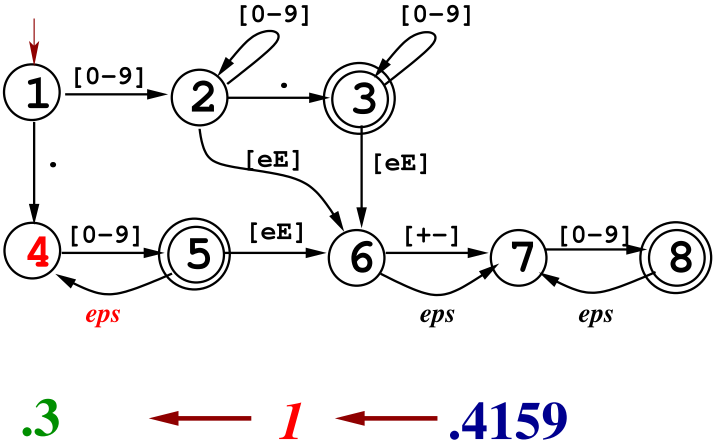
A More Complex Analysis: Float Numbers



.3 ⟵ *eps* ⟵ 1.4159

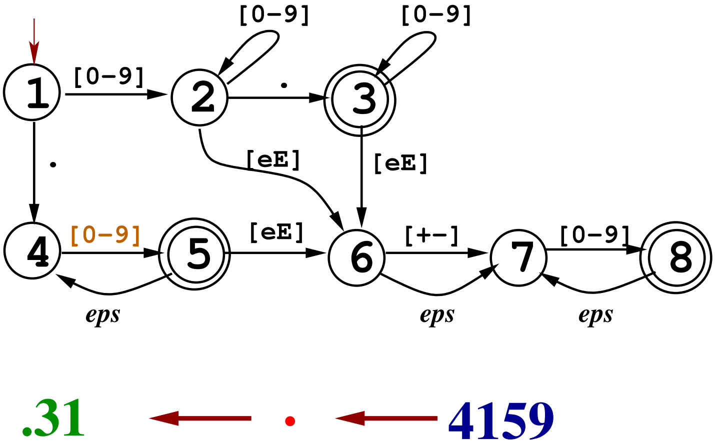
Starting at the pointed state,
transitions to new state possibly reading input.

A More Complex Analysis: Float Numbers



Starting at the pointed state,
transitions to new state possibly reading input.

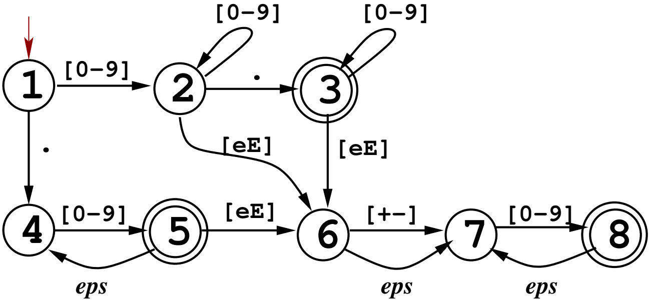
A More Complex Analysis: Float Numbers



If no transition: stuck ⇒ input refused!

If at end of input: check if state is accepting!

Float Numbers: NFA Formalization



.31.4159

- $S = \{1, \ldots, 8\}, \sum = \{0, 1, \ldots, 9, ., e, E\}, s_0 = 1\}, \; F = \{3, 5, 8\}$
- $T = \{1^d 2,\ 1^{.}4,\ 2^d 2,\ 2^{.}3,\ 2^e 6,\ 2^E 6,\ 3^d 3,\ 3^e 6,\ 3^E 6,$
  $4^d 5,\ 5^\epsilon 4,\ 5^e 6,\ 5^E 6,\ 6^+ 7,\ 6^- 7,\ 6^\epsilon 7,\ 7^d 8,\ 8^\epsilon 7\},$

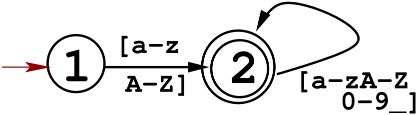
  where $d \in \{0, \ldots, 9\}$. **Picture sufficient as definition.**

Other examples / Exercise

Identifiers:

- $\sum$: letters, digits, underscore,
- starts with a letter or underscore,and follows with any number of letters, digits and underscores:

RE: `[a-zA-Z_] [a-zA-Z0-9_]*`



Binary Numbers

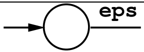
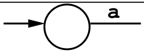
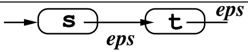
- without leading zeros
- $\sum = 0, 1,\ S = \{0, 1, 2\}$
- $s_0 = 0,\ F = \{1, 2\}$
- $T = \{0^0 1, 0^1 2, 2^0 2, 2^1 2\}$

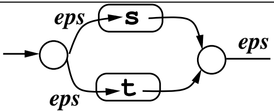
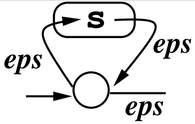
RE: $0 \mid 1\ [01]^*$

NFA: ?

Converting a Regular Expression to an NFA

- Define an NFA fragment for every regular expression constructor. Fragments have exactly one entry (arrow) and exit (line).
- Exit line is labeled either by ϵ or a char $\in \sum$.
- Fragment composition follows expression composition
  A single final state is added at the end of the construction.

RE	Fragment
ϵ	
a	
$s \cdot t$	

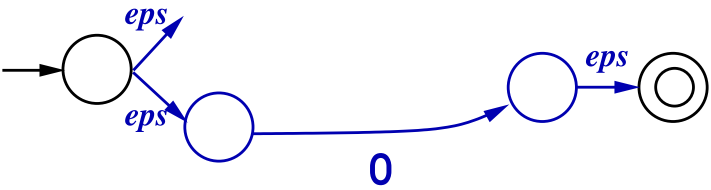
RE	Fragment
$s \mid t$	
s^*	

Construction Example: Binary Numbers Example

`0 | 1 (0 | 1)*`

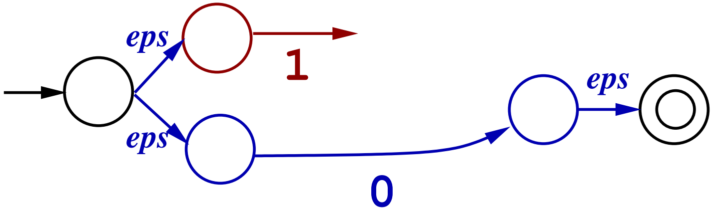
Construction Example: Binary Numbers Example

```
0 | 1 (0 | 1)*
```



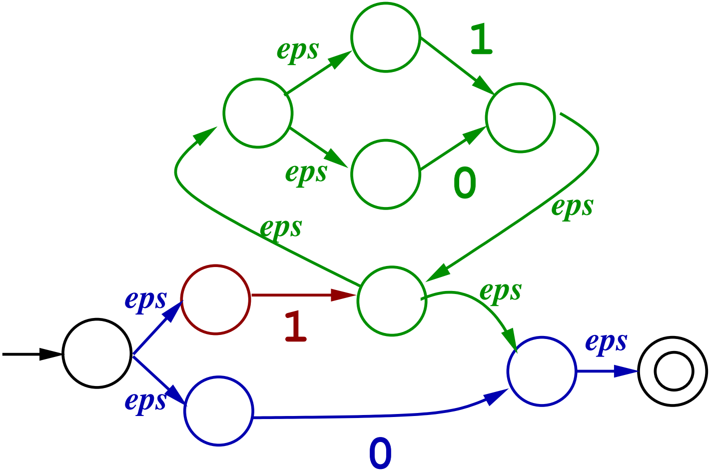
Construction Example: Binary Numbers Example

0 | 1 (0 | 1)*

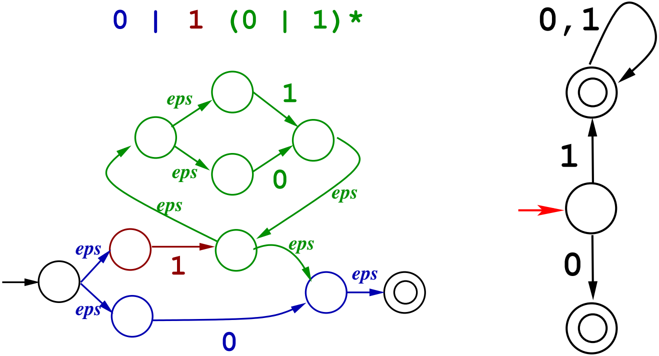


Construction Example: Binary Numbers Example

0 | 1 (0 | 1)*



Construction Example: Binary Numbers Example



Non-determinism is undesired:

- Many ϵ transitions (branch and exit for alternatives),
- Multiple choices, e.g., in which an input can be accepted.

Converting NFA to Deterministic FA (DFA)

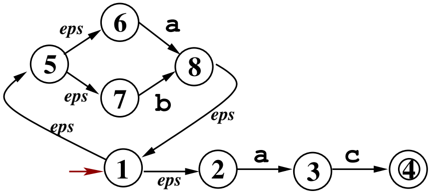
Definition (DFA)

Let $\sum$ be an alphabet of (input) characters. A DFA consists of

- *An alphabet $\sum$ of input characters,*
- *A finite set S of states:*
- *A start (initial) state $s_0 \in S$*
- *A set of final (accepting) states $F \subseteq S$, and*
- *A function* `move` : $S \times \sum \longrightarrow S$ *describing state transitions.*

- Meaning of `move`(s_1,c)=s_2: if in s_1 with input a, go to s_2.
- Note: `move`(s_1,c) can be undefined, i.e., is a partial function.
- No ϵ transitions, no 2 identically labeled transitions from a state.
- **At most one transition possible from any state**, uniquely identified by current state & input char.

Converting an NFA to DFA: Idea



- States 1,2,5,6,7 are reachable from state 1.
- With input a the NFA can go to state 3 and 8.
- On input b only state 8 possible.
- States 1,2,5,6,7 reachable from state 8.
- If in state 3, the NFA can go to state 4 on input c (otherwise nowhere).

In what states can I go at start, on an a, on a b, and on a c?

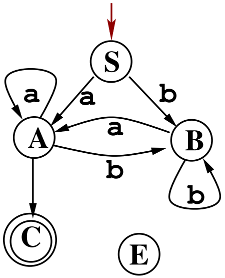
S: {1,2,5,6,7}

A: {1,2,3,5,6,7,8}

B: {1,2,5,6,7,8}

C: {4}

E: $\emptyset$



Epsilon-Closure and Solving Set Equations

Definition (ϵ-Closure)

Let $N = (S, \sum, s_0, F, T)$ a NFA, and $M \subseteq S$ a set of states. The ϵ-closure of M, written $\hat{\epsilon}(M)$ contains all states reachable from states in M by ϵ transitions. It is recursively defined as:

1. $M \subseteq \hat{\epsilon}(M)$
2. *If* $s \in \hat{\epsilon}(M)$ *then* $\{s' \mid s^{\epsilon}s' \in T\} \subseteq \hat{\epsilon}(M)$.

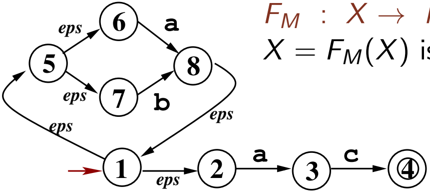
$\hat{\epsilon}(M)$ is the smallest subset of S that fulfills these conditions, or As a Set Equation: $X = M \cup \{s' \mid \exists s \in X \text{ s.t. } s^{\epsilon}s' \in T\}$

Solve this equation by computing a **fixed point** of F_M:

$$F_M : X \to M \cup \{s' \mid \exists s \in X \text{ s.th. } s^{\epsilon}s' \in T\}$$

- F is monotonic: $X \subseteq Y \Rightarrow F(X) \subseteq F(Y)$
- Start by $X_0 = \emptyset$ and compute $X_i = F(X_{i-1})$ ... until $X_n = F(X_n)$
- We have $\emptyset \subseteq F(X_0) \subseteq F(X_1) \subseteq \ldots \subseteq F(X_{n-1}) = X_n = F(X_n)$, i.e., fixed-point reached because the set of states is finite!

Epsilon-Closure Example



$F_M \;:\; X \to \; M \;\cup\; \{s' \mid \exists s \in X \; s.th. \; s^{\epsilon} s' \in T\}$

$X = F_M(X)$ is the epsilon-closure of M, $\hat{\epsilon}(M)$

Starting with $X_0 = \emptyset$, compute:

$X_i = F_M(X_{i-1}) = F^i_M(\emptyset)$

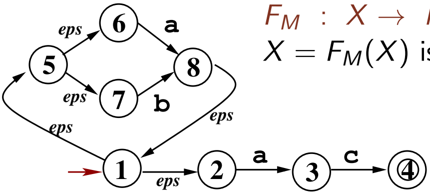
... until $X_n = F(X_n)$!

$\hat{\epsilon}(\emptyset)$	$=$	$\emptyset$
$\hat{\epsilon}(\{1\})$ $\hat{\epsilon}(\{8\})$	$=$ $=$	? ?
		what else is needed?

You may use that F_M is distributive, i.e., $F(X \cup Y) = F(X) \cup F(Y)$, e.g., $F_{\{1\}}(\{1,2,5\}) = F_{\{1\}}(\{1\}) \;\cup\; F_{\{1\}}(\{2,5\})$ and if we already computed $F_{\{1\}}(\{1\})$ we need not recompute it again!

Leads to a worklist algorithm based on marking processed nodes!

Epsilon-Closure Example



$F_M \; : \; X \to \; M \; \cup \; \{s' \mid \exists s \in X \; s.th. \; s^{\epsilon} s' \in T\}$

$X = F_M(X)$ is the epsilon-closure of M, $\hat{\epsilon}(M)$

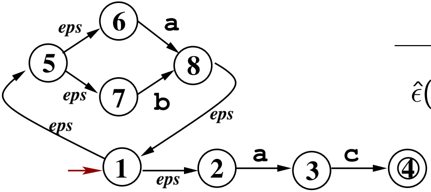
Starting with $X_0 = \emptyset$, compute:

$X_i = F_M(X_{i-1}) = F_M^i(\emptyset)$

… until $X_n = F(X_n)$!

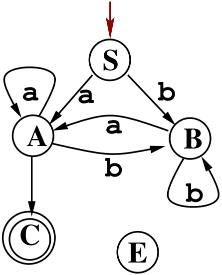
$\hat{\epsilon}(\emptyset)$	$=$	$\emptyset$
$\hat{\epsilon}(\{1\})$	$=$	$\{1,2,5,6,7\}$ $X_1 = 1, \; F(X_1) = X_2 = \{1,2,5\}$ $F(X_2) = X_3 = \{1,2,5,6,7\} = F(X_2) = \hat{\epsilon}(\{1\})$
$\hat{\epsilon}(\{8\})$	$=$	$\{1,2,5,6,7,8\}$
$\hat{\epsilon}(\{3,8\})$	$=$	$\{1,2,3,5,6,7,8\}$
$\hat{\epsilon}(\{4\})$	$=$	$\{4\}$

Epsilon-Closure Example



$$
\begin{array}{rcll}
\hat{\epsilon}(\{1\}) & = & S: & \{1,2,5,6,7\} \\
\hat{\epsilon}(\{3,8\}) & = & A: & \{1,2,3,5,6,7,8\} \\
\hat{\epsilon}(\{8\}) & = & B: & \{1,2,5,6,7,8\} \\
\hat{\epsilon}(\{4\}) & = & C: & \{4\} \\
\hat{\epsilon}(\emptyset) & = & E: & \emptyset
\end{array}
$$

$\texttt{move}(s^d, c) = \hat{\epsilon}(\ \{t \mid s \in s^d \text{ and } s^c t \in T\}\ ),$
$s^d \in DFA,\ s, t \in NFA$



Theorem: Subset Construction

DFA uses same $\sum$; each DFA state is a subset of NFA states.

Definition (ϵ-Closure)

Let $N = (S, \sum, s_0, F, T)$ a given NFA.
Define a DFA $D = (S^d, \sum, s_0^d, F^d, \mathtt{move})$ as follows:

- $S^d = \mathbb{P}(S)$, *i.e., sets of all subsets of* S.
- $s_0^d = \hat{\epsilon}(\{s_0\})$,
- $F^d = \{s^d \in S^d \mid s^d \cap F \neq \emptyset\}$
- $\mathtt{move}(s^d, c) = \hat{\epsilon}(\ \{t \mid s \in s^d \text{ and } s^c t \in T\}\ )$

$\hat{\epsilon}(M)$ *is the smallest subset of* S *that fulfills these conditions, or*
As a Set Equation: $X \;=\; M \;\cup\; \{s' \mid \exists s \in X \text{ s.th. } s^{\epsilon} s' \in T\}$

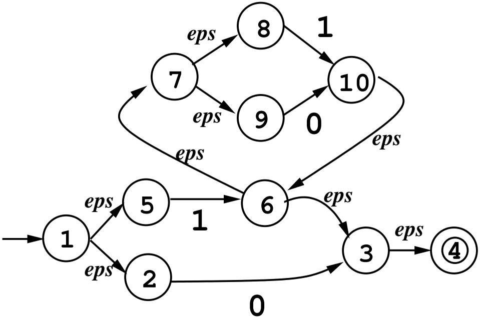
This defines a DFA, which accepts the same language as the NFA named N!

NFA to DFA Tradeoff

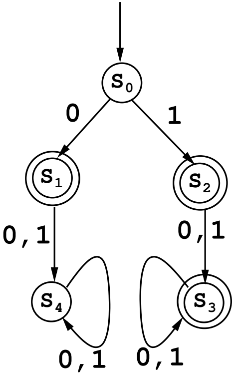
- Size of DFA may reach (rarely) $2^n - 1$ for a n-state NFA.
- DFA can be run in $k \cdot |v|$, with k small constant, and $|v|$ length of input,
- NFA can be run in $c \cdot |n| \cdot |v|$, $c > k$ constant.
- DFA much faster $\Rightarrow$ only when size is a problem consider using a NFA directly.

Motivation for Minimization:

0 | 1 (0 | 1)*

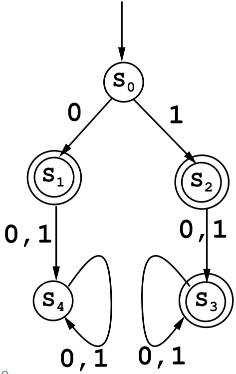


$S_0 : \{1, 2, 5\}$
$S_1 : \{3, 4\}$
$S_2 : \{6, 3, 7, 4, 8, 9\}$
$S_3 : \{10, 6, 3, 7, 4, 8, 9\}$
$S_4 : \emptyset$ (error)



DFA Minimization

- The DFA we obtain from NFA construction are not *minimal*
- Sometimes they contain 'superfluous' states.
- Many lexer-generators perform minimisation
- Property: Any regular language has a unique minimal DFA.



- Dead states: states that cannot lead to a final state with any input.
- States that have identical transitions as others
- Equivalent states: lead to the same outcome (acceptance/rejection) for any input.
- Which is superfluous?
- s_4 is dead! s_2 and s_3 are equivalent!

DFA Minimization Algorithm

Algorithm (DFA Minimization)

Let $D = (S, \sum, s_0, F, \mathtt{move})$ a DFA. We assume `move` total.
Determine state equivalence for a minimised DFA as follows:

1. *Start with two unmarked groups, F and $S \backslash F$.*
2. *While there exist unmarked groups do:*
   a. *pick an unmarked group G*
   b. *for all $a \in \sum$ check $\forall s \in G$ to which group `move(s,a)` leads to.*
   c. *if for any input `a`, all transitions lead to the same group, then mark the group*
   d. *Otherwise split the group into maximal subgroups that lead to the same group on transitions and unmark ALL groups!*
3. *Repeat from 2 until all the groups are marked.*

The resulting groups contain equivalent states!

Minimization Example

Example

Minimization Algorithm assumes either that:

- there are no dead states in the DFA, or
- that the `move` function is total.

In the presence of both, the minimization might fail!

What if `move` is not total?

Make it total by adding a default dead state!
All dead states are equivalent now and can be removed!
The resulting groups contain equivalent states!

Lexer

Results so far: Is an input *w* described by regular expression α?
Decision problem: for $w \in \sum^*$, is *w* in the language described by $\alpha \in RE(\sum)$?

But how to recognize a whole sequence of characters?

```
// My program\n val result =\n  let val x = 10 :: 20 :: 0x30 :: []\n  in
List.map (fn a => 2 * 2 * a) x\n  end
```

$\downarrow$

```
Keywd_Val, Id "result", Equal, Keywd_Let, Keywd_Val, Id "x", Equal, Int 10,
Op_Cons, Int 20, Op_Cons, Int 48, Op_Cons, LBracket, RBracket, Keywd_in,
Id "List", Dot, Id "map", LParen, Keywd_fn Id "a", Arrow, Int 2, Multiply,
Int 2, Multiply, Id "a", RParen, Id "x", Keywd_end
```

- Recognize prefixes of input as tokens,
- Restart on remaining input after recognizing something,
- Often, Several decompositions of the input are possible.

Example The string if21 can be split in several different ways.

Principles: Longest and First Match

Definition (Principle of Longest Match)

A lexical analyser usually outputs the token that consumes the longest part of the input

Why?Important when reading identifiers and numbers!

Definition (Principle of First Match)

Tokens are usually prioritized, so the lexer can decide which token to recognize if two tokens are possible for the same input.

Why?Important when reading keywords, otherwise might be recognized as identifiers...

Define a combined DFA with prioritized final states, backtrack.

Construction of Automaton

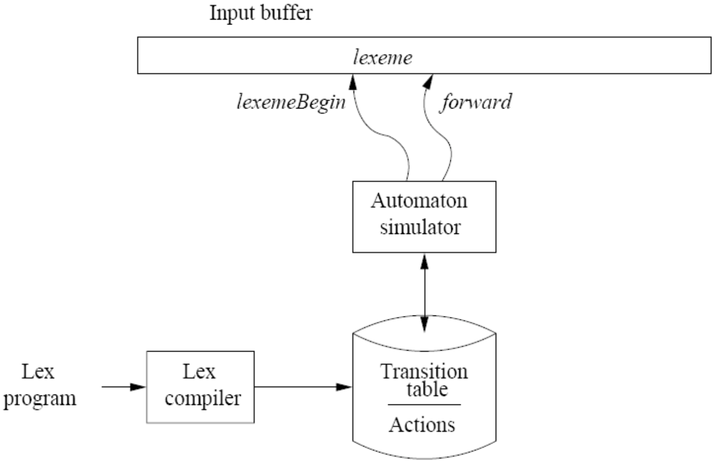
1. define an NFA for each token class,
2. mark final states in each NFA with the token name,
3. union the NFAs using choice, new start and ϵ transitions
4. construct a small combined DFA, using subset construction and minimization. Prioritize token classes in final DFA states to decide what to recognize.

Processing Input with the automaton:

1. start the DFA in normal mode,
2. when a final state is reached, save it and continue reading,
3. In read-ahead mode:
   - buffer all input when reading,
   - when reaching a new final state clear the buffer,
   - IF end-of-input or DFA stuck then output the last final state, restore input from buffer, continue lexing from the last final state.
4. Restart in normal mode until inputs ends.

Lexer Generator

A program that takes a set of token definitions and generate a lexer is called a Lexer Generator.



More About Regular Languages (RLs)

- by definition RLs described by regular expressions, but also by DFA and NFAs,
- RLs are closed under union, concatenation and unbounded repetition,
- also the complement $\sum^* - L$ is regular (if L is regular)
  - construct DFA for L (make sure `move` is total)
  - change every accepting state to non-accepting and vice-versa.
- less trivially, RLs are closed under intersection and subtraction ($L1 \cap L2 \equiv \overline{\overline{L1} \cup \overline{L2}}$ and $L1 - L2 = L1 \cap \overline{L2}$)

More About Regular Languages (RLs)

- The minimized DFA is uniquely determined!hence two regular expressions are equivalent if their minimized DFA is the same (modulo renaming states)!
- Regular languages are limited: what requires unbounded memory cannot be expressed as a regular language, e.g., unbounded counting:
- the palindrome language `kayak, racecar` is not regular!