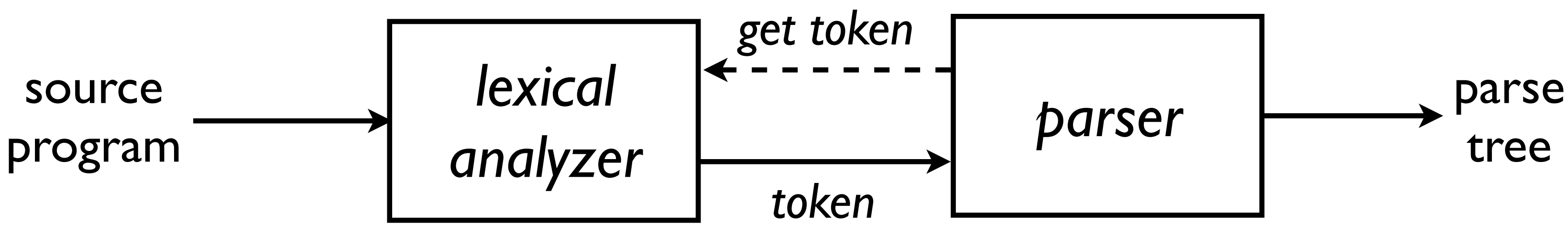
Lesson 2
Lexical Analysis

CS 226/326
Spring 2003

Lexical Analysis

- *Transform source program (a sequence of characters) into a sequence of **tokens**.*



- *Lexical structure is specified using **regular expressions***
- *Secondary tasks*
  1. *discard white space and comments*
  2. *record positional attributes (e.g. char positions, line numbers)*

Example Program

A sample source program in Tiger

```
let
    function g(a:int) = a
in
    g(2,"str")
end
```

What are the tokens?

LET	FUNCTION	ID "g"
LPAREN	ID "a"	COLON
ID "int"	RPAREN	EQ
ID "a"	IN	ID "g"
LPAREN	INT "2"	COMMA
STRING "str"	RPAREN	END

Tokens

Tokens	Text	Description
LET	let	keyword LET
END	end	keyword END
PLUS	+	arithmetic operator
LPAREN	(	punctuation
COLON	:	punctuation
STRING	“str”	string
RPAREN	)	punctuation
INT	46	integer literal
ID	g, a, int	variables, types
EQ	=	
EOF		end of file

Strings

- *Alphabet:* Σ – *a set of basic characters or symbols*
  - finite or infinite, but we will only be concerned with finite Σ
  - e.g. printable Ascii characters
- *Strings:* Σ^* – *finite sequences of symbols from* Σ
  - e.g. ϵ (the empty string), `abc`, `*?x_2`
- *Language:* $L \subseteq \Sigma^*$ – *a set of strings*
  - e.g. L = {ϵ, `a`, `aa`, `aaa`, ...}
- *Concatenation:* $s \cdot t$ ≡ *concatenation of strings s and t*
  - e.g. `abc` · `xy` = `abcxy`
- $\langle \Sigma^*, \cdot, \epsilon \rangle$ *is a semigroup*
- *Product of languages:* $L_1 \cdot L_2 = \{ s\,t \mid s \in L_1 \;\&\; t \in L_2 \}$

Regular Expressions

Regular expressions are a small language for describing languages (i.e. subsets of Σ^*).

Regular expressions are defined by the following *grammar*:

M	::=	**a**	-- *a single symbol* (a $\in$ Σ)
		$M_1 \mid M_2$	-- *alternation*
		$M_1 \;\; M_2$	-- *concatenation (also* M_1M_2 *)*
		ϵ	-- *epsilon*
		M^*	-- *repetition (0 or more times)*

Examples:

$(\mathbf{a} \;\; \mathbf{b}) \mid \epsilon$

$(\mathbf{0} \;\; \mathbf{1})^* \;\; \mathbf{0}$

$\mathbf{b}^*(\mathbf{abb}^*)^*(\mathbf{a}|\epsilon)$

Regular Expressions

The previous forms of regular expressions are adequate, but for convenience we add some redundant forms that could be defined in terms of the basic ones.

M ::= **. . .**

M^+	-- *repetition (1 or more times)*
M?	-- *0 or 1 occurrence of* M
[a-z]	-- *ranges of characters (alternation)*
.	-- *any character other than newline (*`\n`*)*
"**abc**"	-- *literal sequence of characters*

Defs: $M^+ = M\ M^*$

M? = M | ϵ

[a-z] = (**a** | **b** | **c** | ... | **z**)

"**abc**" = **a b c**

Meaning of Regular Expressions

The meaning of regular expressions is given by a function L from regular expressions (re's) to languages (subsets of Σ^*).
L is defined by the equations:

$$L(\mathbf{a}) = \{a\}$$
$$L(M_1 \mid M_2) = L(M_1) \cup L(M_2)$$
$$L(M_1 \; M_2) = L(M_1) \; L(M_2)$$
$$L(\epsilon) = \{\epsilon\}$$
$$L(M^*) = \{\epsilon\} \mid (L(M) \; L(M^*))$$

Examples

$L((\mathbf{a} \; \mathbf{b}) \mid \epsilon) = \{\epsilon, ab\}$

$L((\mathbf{0} \; \mathbf{1})^* \; \mathbf{0}) =$ even binary numbers

$L(\mathbf{b}^*(\mathbf{abb}^*)^*(\mathbf{a}|\epsilon)) =$ strings of a, b with no consecutive a's

Using R.E.s to Define Tokens

Regular expressions are used to define token classes in a specification of lexical structure:

```
if                              (IF)                 -- if keyword
[a-z][a-z0-9]*                  (ID(str))            -- identifier
[0-9]+                          (NUM(str))           -- integer const
([0-9]+"."[0-9]*)|([0-9]*"."[0-9]+)  (REAL(str))
                                                     -- real const
("--"[a-z]*"\n")                (continue())         -- comment
(" "|"\t"|"\n")+                (continue())         -- white space
.                               (error();continue())
                                                     -- error
```

Patterns are matched “top-down”, and the longest match is preferred.

Choosing among Multiple Matches

`if`	`(IF)`	-- *if keyword*
`[a-z][a-z0-9]*`	`(ID(str))`	-- *identifier*

Consider string "`if8`". The initial segment "`if`" matches the first r.e. while the whole string is matches the second r.e. In this case we choose the longest possible match, recognizing the string as an identifier.

Consider "`if 8`". Both the first and second r.e.'s match the initial segment "`if`" and no r.e. matches the entire string (or "`if `" for that matter). In this case we choose the first matching r.e. and recognize the *if* keyword.

Summary: *the longest match is preferred, and ties are resolved in favor of the earliest match.*

Homework Assignment 1

1. Program 1 (p. 10)
   file: prog1.sml

2. Exercise 1.1(a,b,c) (p. 12)
   file: ex1_1.sml

Finite State Machines

The r.e. recognition problem: for *re* M we want to build a machine that scans a string and tells us whether it belongs to *L*(M). Alternatively, in lexical analysis we want to scan a string and find a (longest) initial segment of the string that belongs to *L*(M).

re ⇒ nondeterministic finite automaton (NFA)

⇒ deterministic finite automaton (DFA)

⇒ optimization/simplification of the DFA

⇒ transition table + matching engine

⇒ code for a lexical analyzer

Finite State Machines

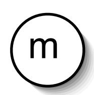
A finite state machine (*finite automaton* or *FA*) over alphabet Σ is a quadruple

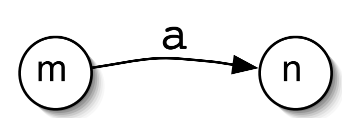
$$M = \langle S, T, i, F \rangle$$

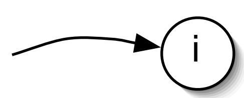
where

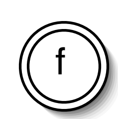
S = a finite set of states (usually represented by numbers)
T = a transition relation: $T \subseteq S \times \Sigma \times S$
i = an initial state $i \in S$
F = a set of final states: $F \subseteq S$

Graphical representations:

$m \in S$: 

$\langle m,a,n \rangle \in T$: 

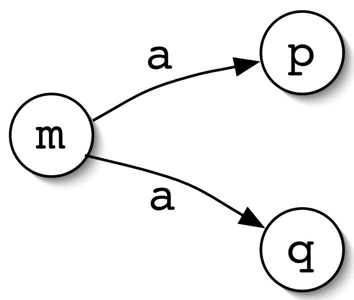
$i \in S$: 

$f \in F$: 

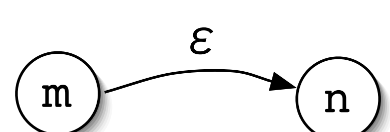
Deterministic and Nondeterministic FA

A finite automata M = ⟨S, T, i, F⟩ is *deterministic* (a DFA) if for each m ∈ S and a ∈ Σ there is at most one n ∈ S such that ⟨m, a, n⟩ ∈ T

Graphically, in a DFA we don't have any situations of the form:

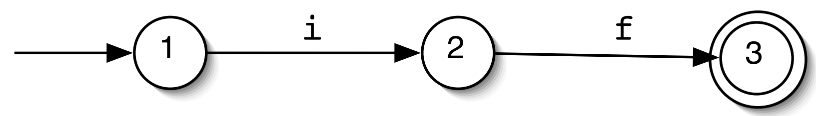


If a FA is not deterministic, it is a nondeterministic FA (an NFA). Nondeterministic automata are also formed by introducing ε transitions -- silent transitions that can be taken without consuming an input symbol.

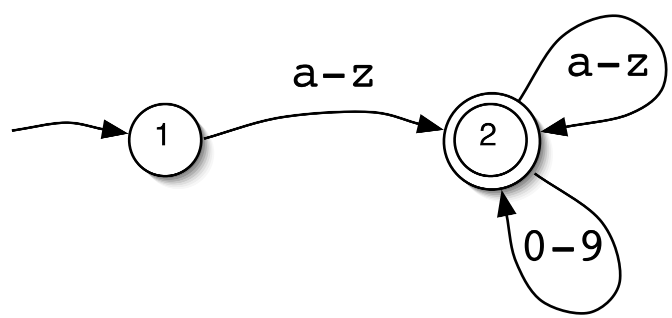


DFAs for Token Classes

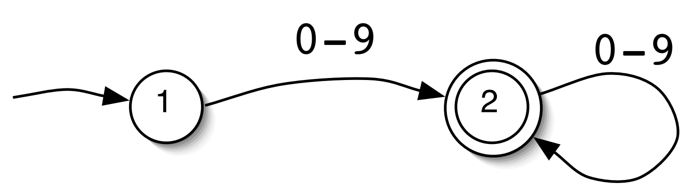
`if` (IF)



`[a-z][a-z0-9]*` (ID(str))

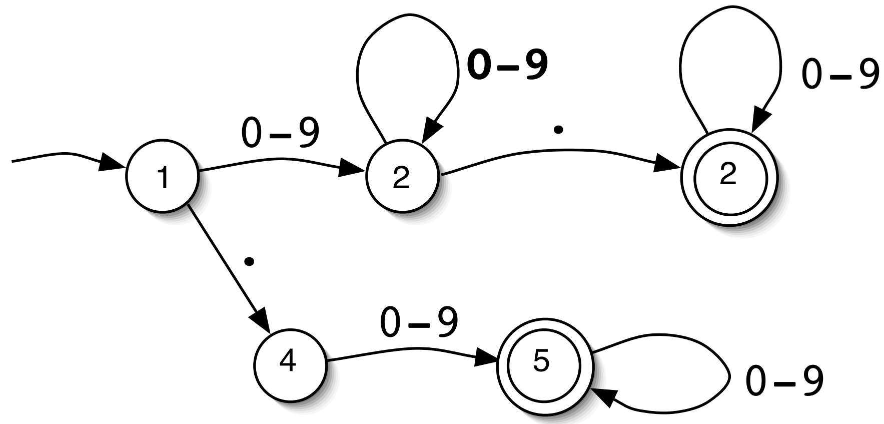


$[0\text{-}9]^+$ (NUM(str))

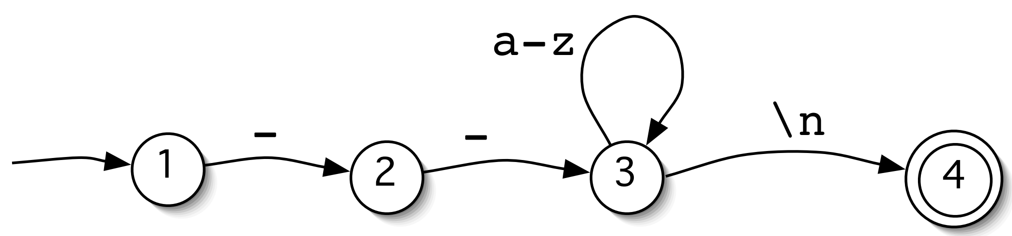


DFAs for Token Classes

([0-9]$^{+}$"."[0-9]*)|([0-9]*"."[0-9]$^{+}$)   (REAL(str))

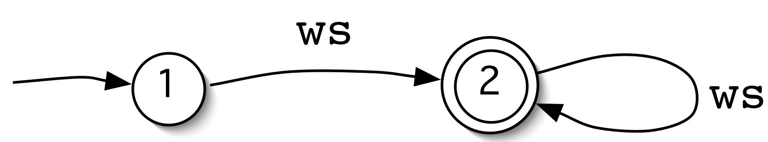


("--"[a-z]*"\n")   (continue())   -- *comment*



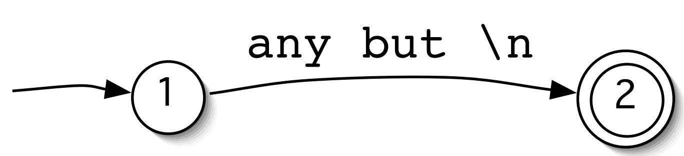
DFAs for Token Classes

`(" "|"\t"|"\n")+` `(continue())` -- *white space*

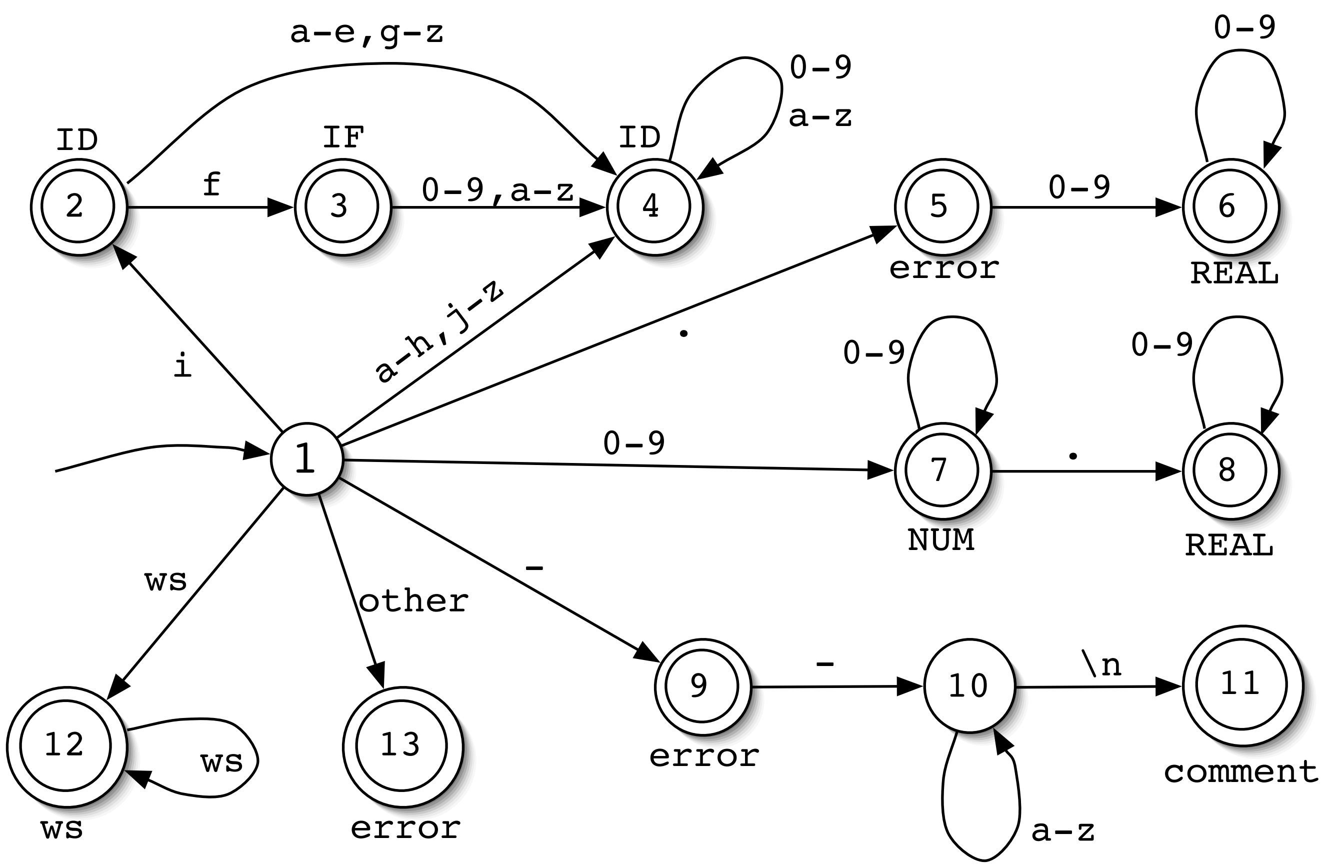


where `ws` *is* `(" "|"\t"|"\n")`

`.` `(error();continue())` -- *error*

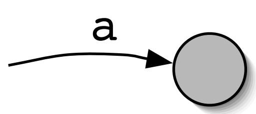


Combined DFA

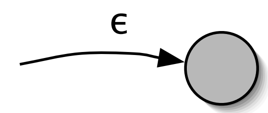


R.E. to NFA

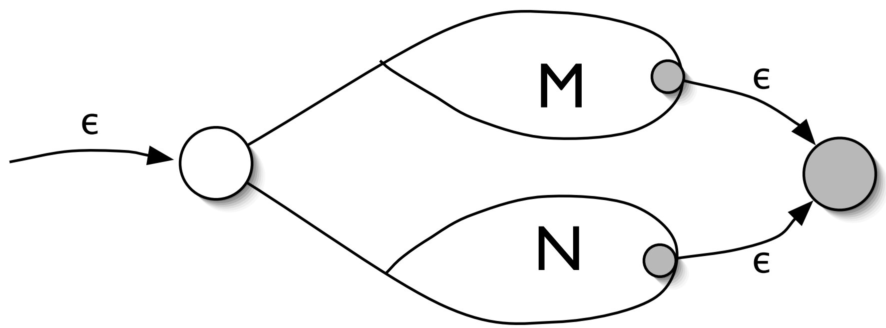
a



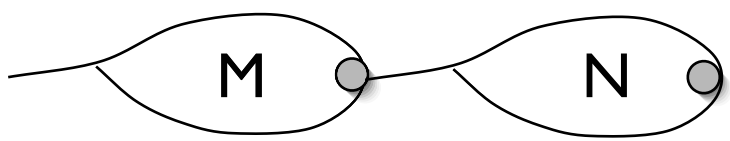
ε



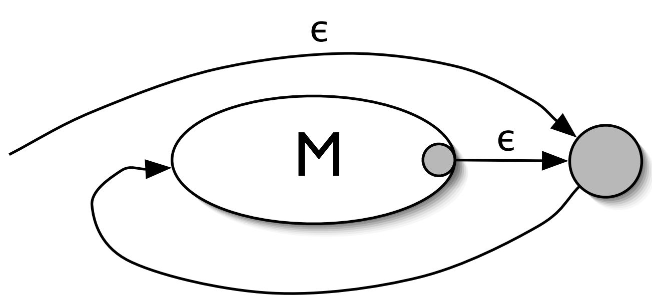
M | N



M N

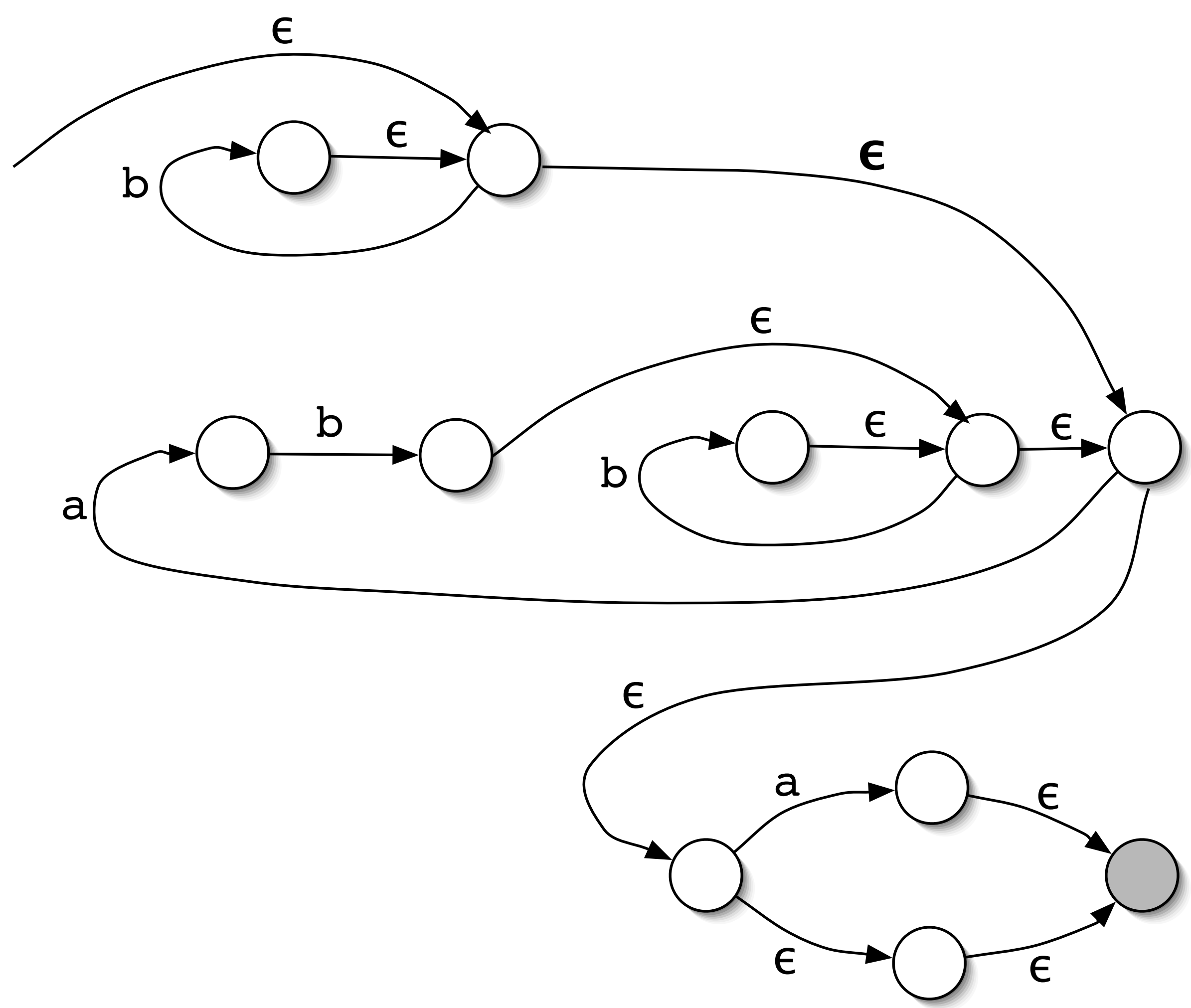


M^*

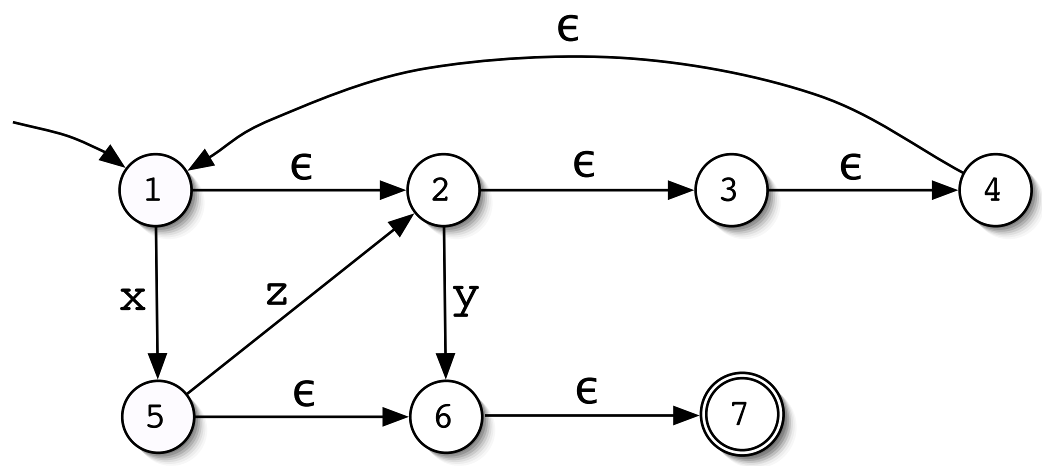


RE to NFA Example

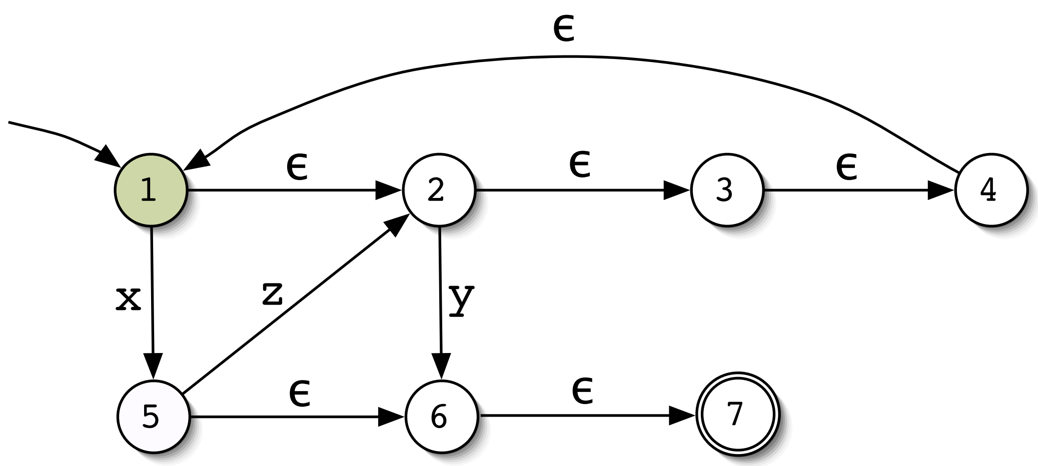
$\mathbf{b^*(abb^*)^*(a|\epsilon)}$

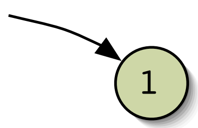


NFA to DFA

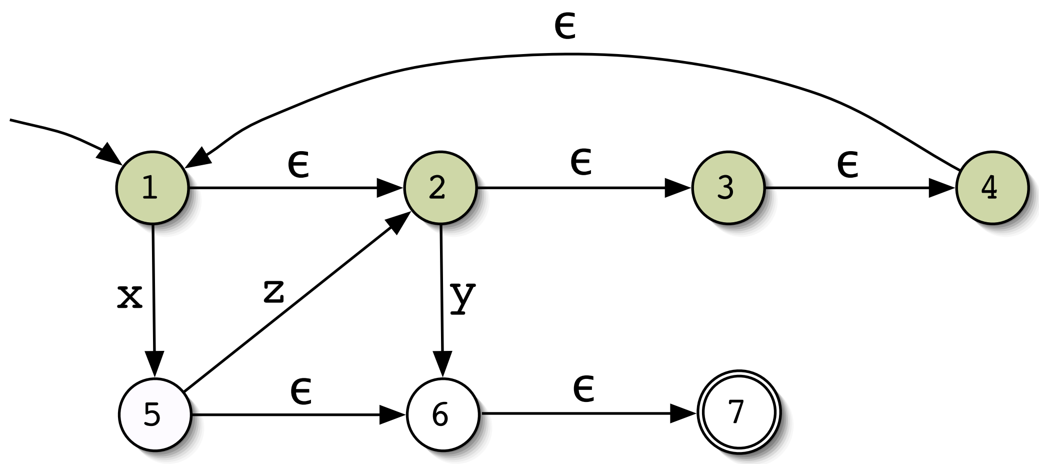


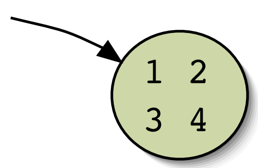
NFA to DFA





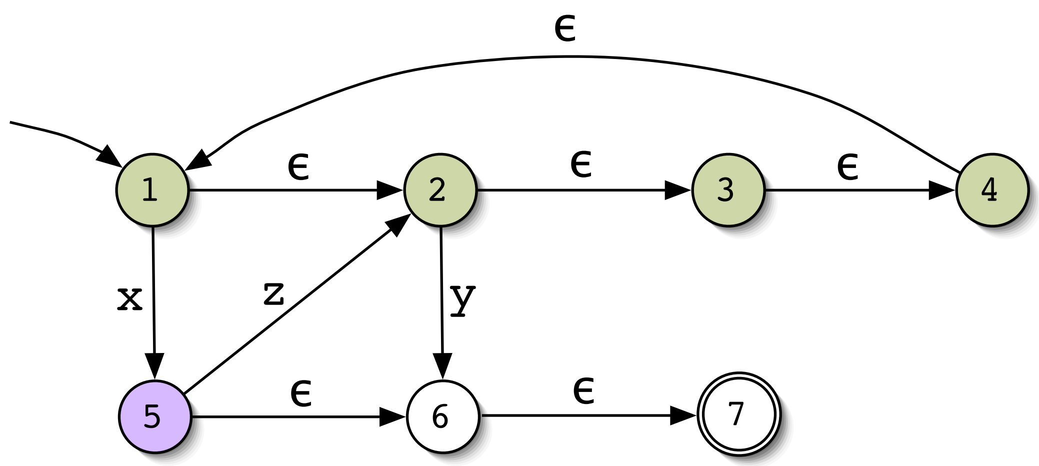
NFA to DFA

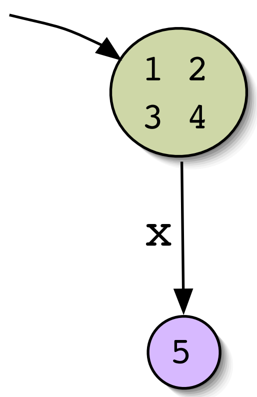




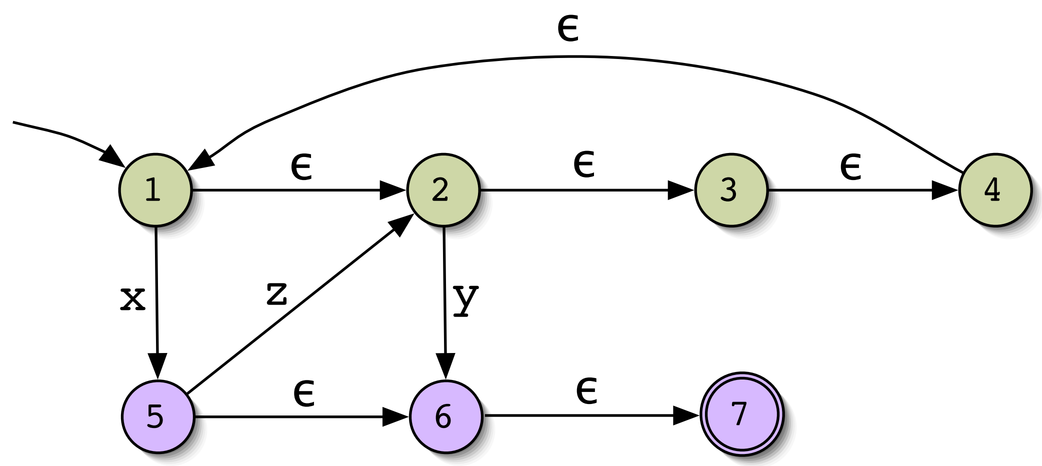
ϵ–closure of 1

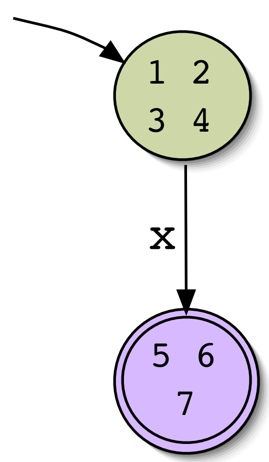
NFA to DFA





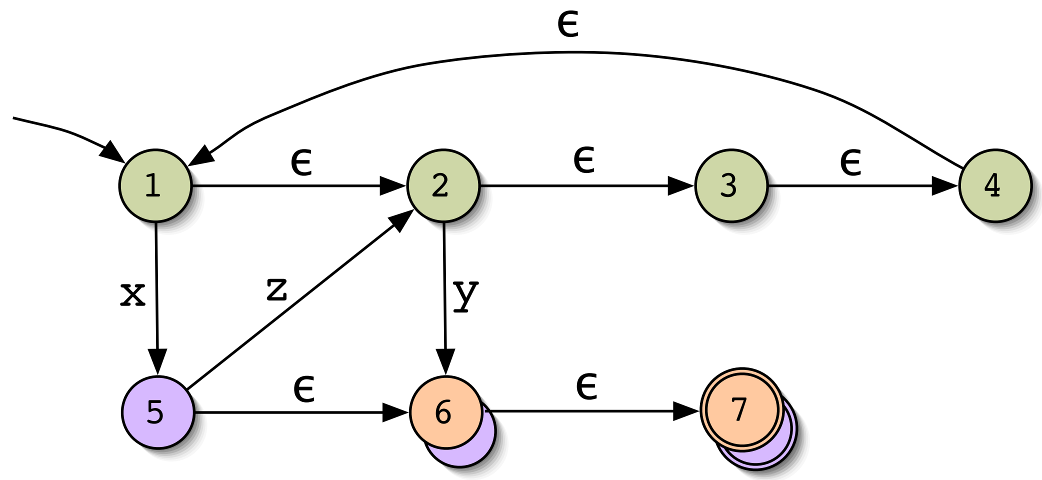
NFA to DFA

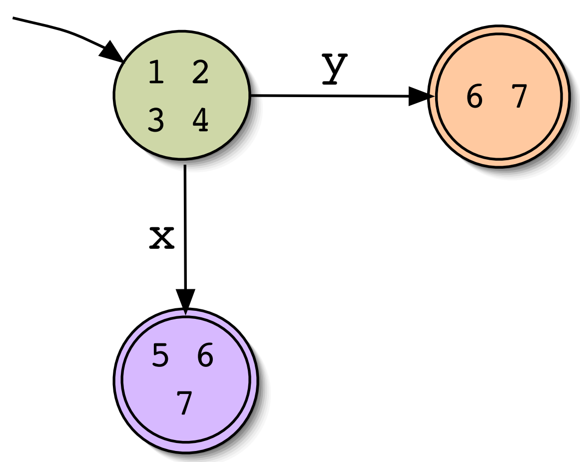




ϵ–closure of 5

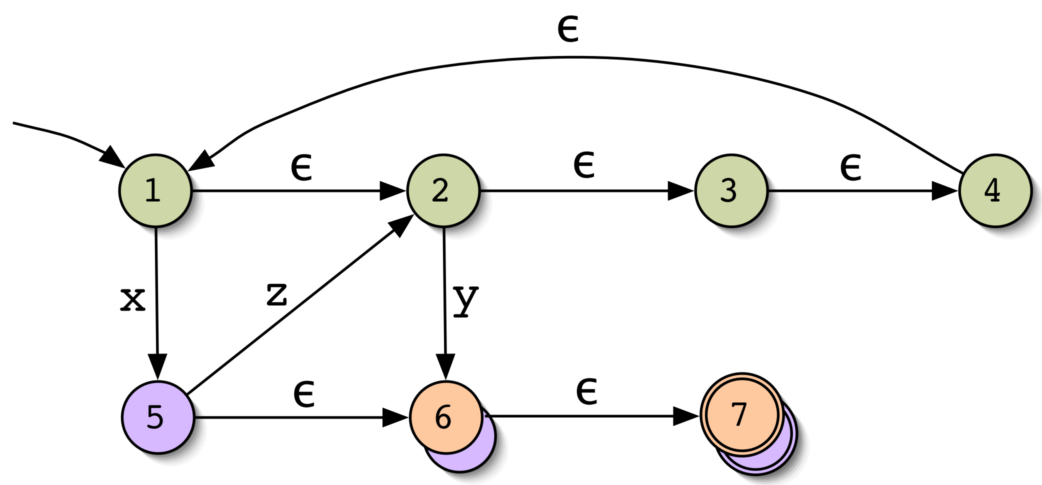
NFA to DFA

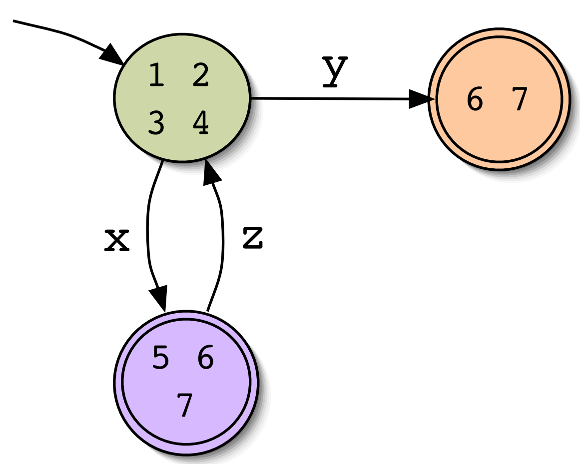




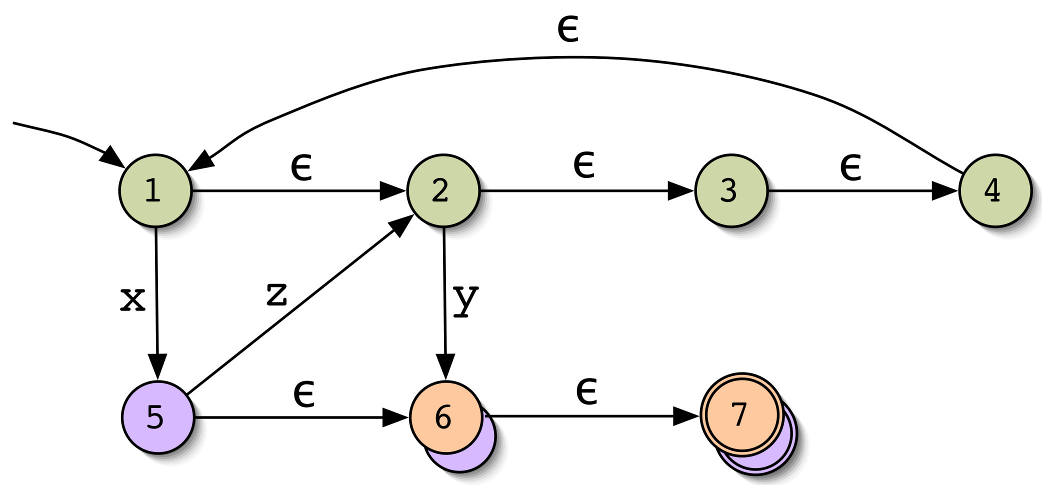
ϵ–closure of 6

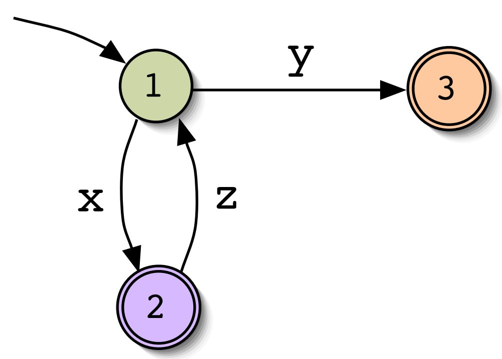
NFA to DFA



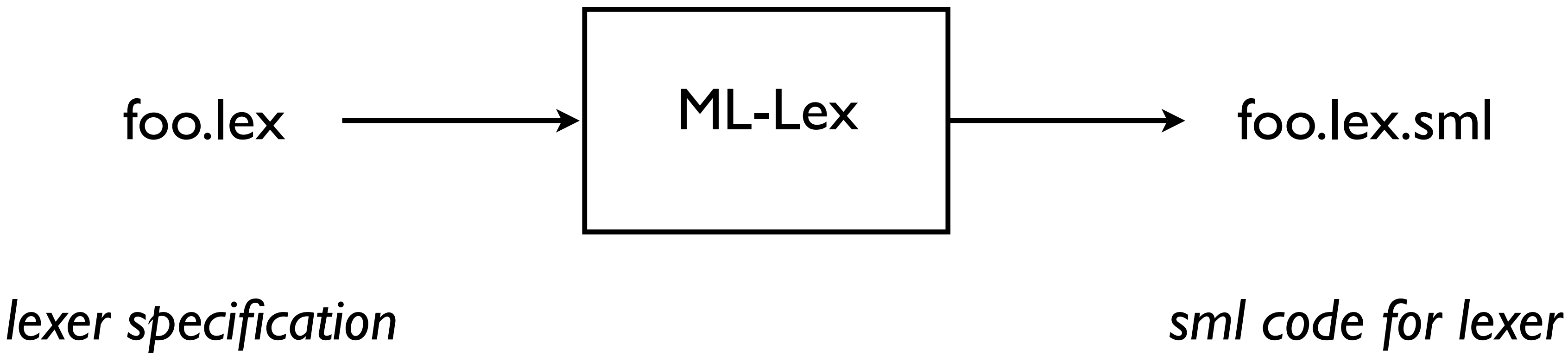


NFA to DFA





ML-Lex



Specification for token values has to be supplied externally, usually in the form of a Tokens module that defines a token type and a set of functions for building tokens of various classes.

An ML-Lex specification

ML Declarations:

```
type lexresult = Tokens.token
fun eof() = Tokens.EOF(0,0)
%%
```

Lex definitions:

```
digits=[0-9]+;
%%
```

Regular Expressions and Actions:

```
if                        => (Tokens.IF(yypos,yypos+2));
[a-z][a-z0-9]*            => (Tokens.ID(yytext,yypos,yypos+size yytext));
{digits}                  => (Tokens.NUM(Int.fromString yytext,yypos,
                                         yypos+size yytext));
({digits}"."[0-9]*)|([0-9]*"."{digits})
                          => (Tokens.REAL(Real.fromString yytext,yypos,
                                          yypos+size yytext));
("--"[a-z]*"\n")          => (continue());
(" "|"\n"|"\t")           => (continue());
.                         => (ErrorMsg.error yypos "illegal character";
                              continue());
```

Variables Defined by ML-Lex

ML-Lex defines several variables:

`lex()`	recursively call the lexer
`continue()`	`same, but with %arg`
`yytext`	the string matched by the current r.e.
`yypos`	character position at start of current r.e. match
`yylineno`	line number at start of match (if command `%count` given)

Defining Tokens

```
(* ML Declaration of a Tokens module (called a structure in ML): *)

structure Tokens =
struct

    type pos = int
    datatype token
      = EOF of pos * pos
      | IF of pos * pos
      | ID of string * pos * pos
      | NUM of int * pos * pos
      | REAL of real * pos * pos
      ...
end  (* structure Tokens *)
```

Start States

Several different lexing automata can be set up using *start states*. Additional start states are commonly used for handling comments and strings.

```
ML decls...
%%
Lex decls...
%s COMMENT
%%
<INITIAL>if             => (Tokens.IF(yypos,yypos+2));
<INITIAL>[a-z]+         => (Tokens.ID(yytext,yypos,
                                      yypos+size yytext));

<INITIAL>"(*"           => (YYBEGIN COMMENT; continue());
<COMMENT>"*)"           => (YYBEGIN INITIAL; continue());
<COMMENT>.              => (continue());
```