

# Scanner

---

Ronghui Gu

Fall 2022

Columbia University

\* Course website: <https://verigu.github.io/4115Fall2022/>

\*\* These slides are borrowed from Prof. Edwards.

# The Big Picture

---

# The First Question

How do we describe/construct a program?

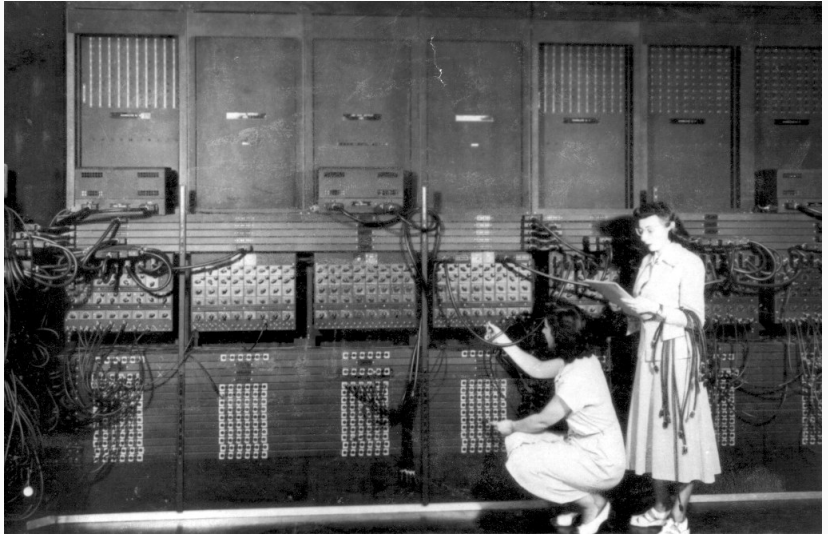
## Use continuously varying values?



Very efficient, but has serious noise issues

Edison Model B Home Cylinder phonograph, 1906

# The ENIAC: Programming with Spaghetti



# Have one symbol per program?

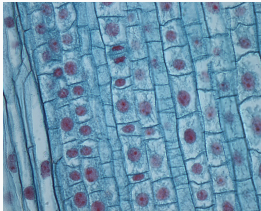
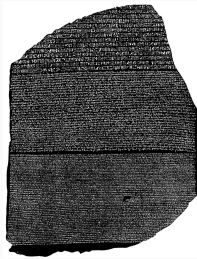
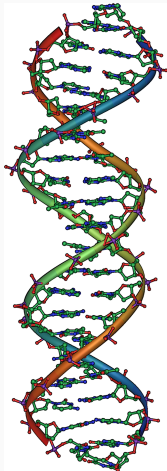


Not so good when there are many, many things






Nippon Typewriter SH-280, 2268 keys

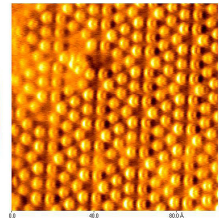
# Solution: Use a Discrete Combinatorial System

Use combinations of a small number of things to represent (exponentially) many different things.



ENGLISH SOUNDS

 chance	 flick	 voice	 book	 look	 eat	 radio
 elephant	 camera	 bird	 ball	 cute	 bag	 phone
 fat	 hut	 car	 lock	 mush	 knife	 cow
 pot	 bottle	 table	 door	 chop	 leaf	 key
 fish	 van	 umbrella	 beaver	 snake	 voice	 shower
 mouse	 airplane	 ring	 house	 light	 ring	 wet



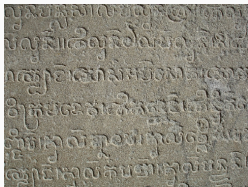
# Every Human Writing System Does This



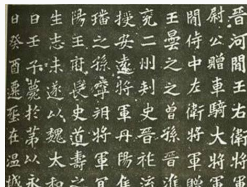
Hieroglyphics (24+)



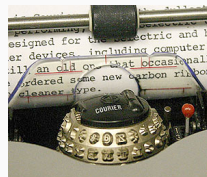
Cuneiform (1000 – 300)



Sanskrit (36)



Chinese (214 – 4000)



IBM Selectric (88-96)

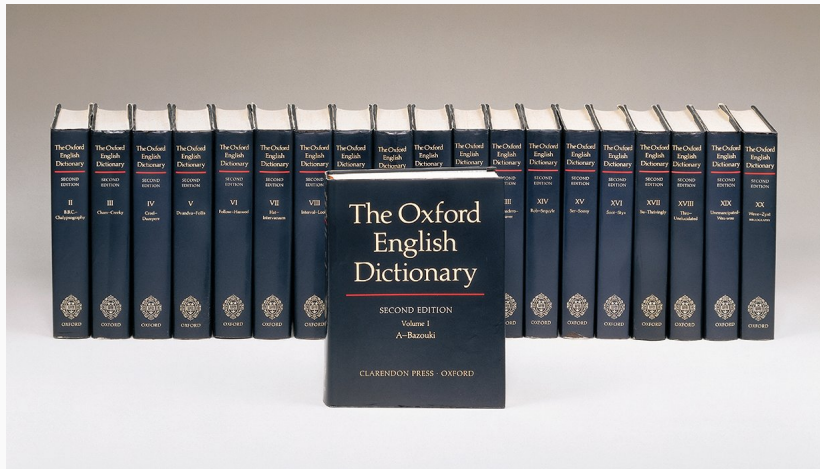




## The Second Question

How do we describe the combinations of a small number of things.

# Just List Them?



Gets annoying for large numbers of combinations

## Just List Them?

## 3 AA—AAAAAAAAAAAAA

AAAAA Budget Moving	16 Wilby Cr.	241-5468
AAAAA Canadian Mail Warehouse	Properties 5399 Talbot/W.	620-1577
1001 Arrowd.		734-0228
24 Jefferson Av.		533-7572
412 Finch St.		734-1267
AAAAA Critter Control		201-7111
AAAAA Critter Control		
100 Burncrest	Univille 4110	410-8727
AAAAA Devco Glass		410-0701
AAAAA Drainsworks Ltd		
Toronto East		422-0501
AAAAA Evening Remodelers		929-6846
AAAAA EPI Mini Storage	555 Threewaydr	247-6294
AAAAA European		962-0323
AAAAA Expert Movers	16 Wilby Cr.	242-7478
AAAAA Express Moving		929-9975
AAAAA Limousine Connection		
AAAAA Mature Escorts		367-5446
AAAAA Move Master		588-4655
AAAAA Neal Professional Moving	Systems 2480 Lawrence Av.	285-3252
AAAAA North American Moving		28-6701
AAAAA Ski Stockings		538-6309
AAAAA Woodbine Window/Storage Ltd		
AAAAA Airtel Glass Service	65 Crockett Dr.	751-4900
AAAAA All Star Movers		239-1587
	603 Evans	538-1958
AAAAA Armstrong Moving		
AAAAA HSL Moving & Storage		233-2477
AAAAA Middup Moving & Storage		253-7290
AAAAA 1 Moving & Storage	60 Grand Blvd.	494-9451
AAAAA Prestige Movers	637 Lansdowne	516-3536
AAAAA South Western Ontario Wildlife	703 Gladstone	533-2623
AAAAA Speedy Moving	Removal	699-4066
A-A-A-A-A Speedy Moving	124 Cromwell	285-6048
AAAAA Auroa	1540 Victoria Park	751-9533
AAAAA Autos	425 Adelaide W.	504-0008
AAAAA California Dreams Escort	855 Athens	363-8676
AAAAA California Dreams Massage	Service	322-3899
AAAAA National Auto Storage	562 Kilpik	503-3833
AAAAA A NightDay		929-9975
AAAAA Strip 'N' Ride		964-7771
AAAAA Automated Door		386-5337
AAAAA California Bed & Bath	Systems 22 Jolland	255-7127
AAAAA California Bed & Bath		321-9822

AAAAAAAAAA CBS Moving  
 130 Lansdowne, 533-7139  
 AAAAAAAAAA Dream Girls, 255-0503  
 AAAAAAAAAA Big Apple Express  
 Service, 466-2767  
 AAAAAAAAAA Accident and  
 Accompanying Injuries/Criminal  
 Practice 1018 FinchW, 663-2211  
 AAAAAAAAAA Accident  
 Accompanying Injuries/Criminal  
 Practice 1018 FinchW, 663-2211  
 AAAAAAAAAA China Blue Escort  
 Service, 323-9522  
 AAAAAAAAAA AABCO Door Co  
 Practice 1018 FinchW, 740-3667  
 AAAAAAAAAA Action Law  
 5233 DundasSW, 253-0888  
 AAAAAAAAAA Alert Auto  
 Glass, 398-4585  
 Or  
 Glass, 399-3410  
 AAAAAAAAAA AMI Campbell Van Lines  
 283-0660  
 AAAAAAAAAA Automobile Auto Glass  
 Hotline, 283-0042  
 AAAAAAAAAA Collins & Greig  
 Cartage Ltd 312 Cornhill, 239-2991  
 AAAAAAAAAA Competition Auto  
 Glass, 223-1292  
 AAAAAAAAAA Competition Auto  
 Glass, 323-0042  
 AAAAAAAAAA Competition Auto  
 Glass, 410-7693  
 AAAAAAAAAA International  
 Services, 693-6848  
 AAAAAAAAAA Jewel Dating&Escort  
 Service, 461-0629  
 AAAAAAAAAA Market  
 Services, 413-4444  
 AAAAAAAAAA Nothing But  
 Glass, 595-1884  
 AAAAAAAAAA On The Wild Side  
 Sensational Female Escort Service 255-1320  
 21 McCall, 667-0574  
 AAAAAAAAAA The Good Life Clubs  
 1125 FinchW, 239-1422  
 1191 Kennedy, 279-7278  
 302 The EastVal, 239-7783  
 1125 FinchW, 667-0740  
 AAAAAAAAAA Affordable And  
 Aggressive Defence 4950 YongeSt, 221-7103  
 AAAAAAAAAA Carport Moving  
 1125 FinchW, 251-4438  
 AAAAAAAAAA Windshields To  
 Go 159 Dwyer, 787-8039  
 AAAAAAAAAA Sunset  
 Escorts, 622-1177  
 AAAAAAAAAA A Best Of The  
 Best, 329-9209  
 AAAAAAAAAA A Billion Times  
 286 RoyalYork, 255-8518

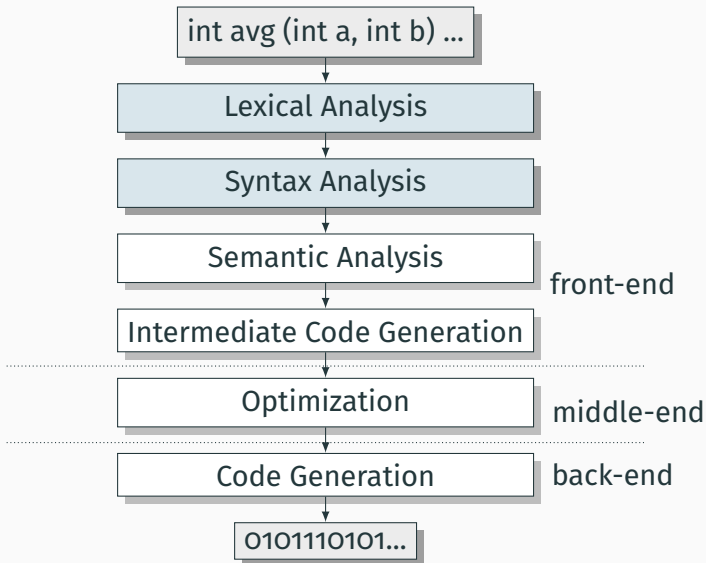
AAAAAAAAAAAA 4

[illegible]

AAAAAAAAAAAAAAAAB Towing 18 Canso. 245-7676  
 AAAAAAAAAAAAAAAAC Robertson Moving/Storage 236 North Queen 620-1212  
 AAAAAAAAAAAAAAAAD Resz, 652-5252  
 AAAAAAAAAAAAAAAAE Accest Law, 784-2020  
 AAAAAAAAAAAAAAAAF Accident  
 Accompanying Injuries/Criminal  
 Practice 1001-458-6632  
 AAAAAAAAAAAAAAAG Claims 2 St.Clair/A 474-2133  
 AAAAAAAAAAAAAAAAH AAAA Account  
 1055 Midfield 294-0750  
 AAAAAAAAAAAAAAAAI AAAA Advant-  
 Edge Door Systems. 222-8322  
 AAAAAAAAAAAAAAAAJ AAAA Adv  
 Executive's Choice. 929-9390  
 AAAAAAAAAAAAAAAAK Automatic Garage Door 64 Clarkson, 785-7820  
 Etobicoke 252-5886  
 AAAAAAAAAAAAAAAL AAAA Cross  
 Alarms 280 Consumers. 494-8777  
 AAAAAAAAAAAAAAM AAAA  
 Mature Escorts 923-3333  
 AAAAAAAAAAAAAAN Professional Express System  
 65 Adelaide/W 504-9111  
 AAAAAAAAAAAAAAO AAAA Sweet  
 Escorts/Dn Yoo 259-3940  
 AAAAAAAAAAAAAAP AAAA AAAA AAAA  
 Marco 1205 Spadina 951-2299  
 AAAAA AAAA AAAA AAAA Domestic  
 Tagliola 1205 St.Clair/A 951-2299  
 AAAAAAAAAAAAAAAQ AAAA Always  
 AAAAAAAAAAAAAAAAR AAAA Touch Of  
 Class Escort Service. 461-8110  
 AAAAAAAAAAAAAAS Apple Auto Glass  
 No Charge-Dial 1 800 506-6055  
 AAAAAAAAAAAAAAT AAAA AAAA AAAA  
 Cardinal Court Lodging 966-4728  
 A A A A A A L U Student Motors . 693-2403  
 A A A A A A B C OOR Door Co.  
 1860 Bnhilf Rd Missauga . Toronto 748-3667  
 A A A A A B S Missuvas  
 643 Lakeshore/Ave. 588-4999  
 A A A A A B C D E F Lockdowns 909 St.Clair/E 822-2255  
 A A A A A B C Movers Inc. 6 Columbus. 535-3413  
 A A A A A B C Best Moving 503-9321  
 A A A A A M O I Moving System  
 905 Midfield/ 294-2323  
 A A A A B Moving 500 Cathedral/. 787-4664  
 A A A A B B E L Lockdowns 966-4728  
 A A A A B C Glass Supply 1111 953-1548  
 A A A A B B D Lockdowns Co  
 1860 Bnhilf Rd Missauga Toronto 748-3667

Can be really redundant

# Scanning and Parsing



# Lexical Analysis

---

# Lexical Analysis (Scanning)

Translate a stream of characters to a stream of tokens



f o o \_ = \_ a + \_ bar ( 0 , \_ 42 , \_ q ) ;

ID EQUALS ID PLUS ID LPAREN NUM COMMA ID  
LPAREN SEMI

Token	Lexemes	Pattern
EQUALS	=	an equals sign
PLUS	+	a plus sign
ID	a foo bar	letter followed by letters or digits
NUM	0 42	one or more digits

# Lexical Analysis

Goal: simplify the job of the parser and reject some wrong programs, e.g.,

```
%#$^#!#%#$
```

is not a C program<sup>†</sup>

# Lexical Analysis

Goal: simplify the job of the parser and reject some wrong programs, e.g.,

```
%#$^#!#%#$
```

is not a C program<sup>†</sup>

Scanners are usually much faster than parsers.



# Lexical Analysis

Goal: simplify the job of the parser and reject some wrong programs, e.g.,

```
%#$^#!#%#$
```

is not a C program<sup>†</sup>

Scanners are usually much faster than parsers.

Discard as many irrelevant details as possible (e.g., whitespace, comments).

# Lexical Analysis

Goal: simplify the job of the parser and reject some wrong programs, e.g.,

```
%#$^#!%#$
```

is not a C program<sup>†</sup>

Scanners are usually much faster than parsers.

Discard as many irrelevant details as possible (e.g., whitespace, comments).

Parser does not care that the identifier is “supercalifragilisticexpialidocious.”

Parser rules are only concerned with tokens.

<sup>†</sup> It is what you type when your head hits the keyboard

# Describing Tokens

**Alphabet:** A finite set of symbols

Examples:  $\{ 0, 1 \}$ ,  $\{ A, B, C, \dots, Z \}$ , ASCII, Unicode

# Describing Tokens

**Alphabet:** A finite set of symbols

Examples:  $\{ 0, 1 \}$ ,  $\{ A, B, C, \dots, Z \}$ , ASCII, Unicode

**String:** A finite sequence of symbols from an alphabet

Examples:  $\epsilon$  (the empty string), Ronghui,  $\alpha\beta\gamma$

# Describing Tokens

**Alphabet:** A finite set of symbols

Examples:  $\{ 0, 1 \}$ ,  $\{ A, B, C, \dots, Z \}$ , ASCII, Unicode

**String:** A finite sequence of symbols from an alphabet

Examples:  $\epsilon$  (the empty string), Ronghui,  $\alpha\beta\gamma$

**Language:** A set of strings over an alphabet

Examples:  $\emptyset$  (the empty language),  $\{ 1, 11, 111, 1111 \}$ , all English words, strings that start with a letter followed by any sequence of letters and digits

## Operations on Languages

Let  $L = \{ \epsilon, \text{wo} \}$ ,  $M = \{ \text{man, men} \}$

**Concatenation:** Strings from one followed by the other

$LM = \{ \text{man, men, woman, women} \}$

# Operations on Languages

Let  $L = \{ \epsilon, \text{wo} \}$ ,  $M = \{ \text{man}, \text{men} \}$

**Concatenation:** Strings from one followed by the other

$LM = \{ \text{man}, \text{men}, \text{woman}, \text{women} \}$

**Union:** All strings from each language

$L \cup M = \{ \epsilon, \text{wo}, \text{man}, \text{men} \}$

# Operations on Languages

Let  $L = \{ \epsilon, \text{wo} \}$ ,  $M = \{ \text{man}, \text{men} \}$

**Concatenation:** Strings from one followed by the other

$LM = \{ \text{man}, \text{men}, \text{woman}, \text{women} \}$

**Union:** All strings from each language

$L \cup M = \{ \epsilon, \text{wo}, \text{man}, \text{men} \}$

**Kleene Closure:** Zero or more concatenations

$M^* = \{ \epsilon \} \cup M \cup MM \cup MMM \dots =$

$\{ \epsilon, \text{man}, \text{men}, \text{manman}, \text{manmen}, \text{menman}, \text{menmen},$   
 $\text{manmanman}, \text{manmanmen}, \text{manmenman}, \dots \}$



## Regular Expressions over an Alphabet $\Sigma$

A standard way to express languages for tokens.

1.  $\epsilon$  is a regular expression that denotes  $\{\epsilon\}$
2. If  $a \in \Sigma$ ,  $a$  is an RE that denotes  $\{a\}$
3. If  $r$  and  $s$  denote languages  $L(r)$  and  $L(s)$ ,

## Regular Expressions over an Alphabet $\Sigma$

A standard way to express languages for tokens.

1.  $\epsilon$  is a regular expression that denotes  $\{\epsilon\}$
2. If  $a \in \Sigma$ ,  $a$  is an RE that denotes  $\{a\}$
3. If  $r$  and  $s$  denote languages  $L(r)$  and  $L(s)$ ,

$(r) \mid (s)$  denotes  $L(r) \cup L(s)$

# Regular Expressions over an Alphabet $\Sigma$

A standard way to express languages for tokens.

1.  $\epsilon$  is a regular expression that denotes  $\{\epsilon\}$
2. If  $a \in \Sigma$ ,  $a$  is an RE that denotes  $\{a\}$
3. If  $r$  and  $s$  denote languages  $L(r)$  and  $L(s)$ ,

$(r) \mid (s)$  denotes  $L(r) \cup L(s)$

$(r)(s)$   $\{tu : t \in L(r), u \in L(s)\}$

## Regular Expressions over an Alphabet $\Sigma$

A standard way to express languages for tokens.

1.  $\epsilon$  is a regular expression that denotes  $\{\epsilon\}$
2. If  $a \in \Sigma$ ,  $a$  is an RE that denotes  $\{a\}$
3. If  $r$  and  $s$  denote languages  $L(r)$  and  $L(s)$ ,

$(r) \mid (s)$  denotes  $L(r) \cup L(s)$

$(r)(s)$   $\{tu : t \in L(r), u \in L(s)\}$

$(r)^*$   $\cup_{i=0}^{\infty} L(r)^i$

where  $L(r)^0 = \{\epsilon\}$

and  $L(r)^i = L(r)L(r)^{i-1}$

# Regular Expression Examples

$$\Sigma = \{a, b\}$$

Regexp.	Language
$a \mid b$	$\{a, b\}$
$(a \mid b)(a \mid b)$	

# Regular Expression Examples

$$\Sigma = \{a, b\}$$

Regexp.	Language
$a \mid b$	$\{a, b\}$
$(a \mid b)(a \mid b)$	$\{aa, ab, ba, bb\}$
$(a \mid b)^*$	

# Regular Expression Examples

$$\Sigma = \{a, b\}$$

Regexp.	Language
$a \mid b$	$\{a, b\}$
$(a \mid b)(a \mid b)$	$\{aa, ab, ba, bb\}$
$(a \mid b)^*$	$\{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, \dots\}$
$a \mid a^*b$	$\{a, b, ab, aab, aaab, aaaab, \dots\}$

## Specifying Tokens with REs

ID: letter followed by letters or digits

Typical choice:  $\Sigma = \text{ASCII characters, i.e.,}$   
 $\{\_, !, ", \#, \$, \dots, 0, 1, \dots, 9, \dots, A, \dots, Z, \dots, \sim\}$

**letters:**  $A \mid B \mid \dots \mid Z \mid a \mid \dots \mid z$

**digits:**  $0 \mid 1 \mid \dots \mid 9$

**identifier:**



# Specifying Tokens with REs

ID: letter followed by letters or digits

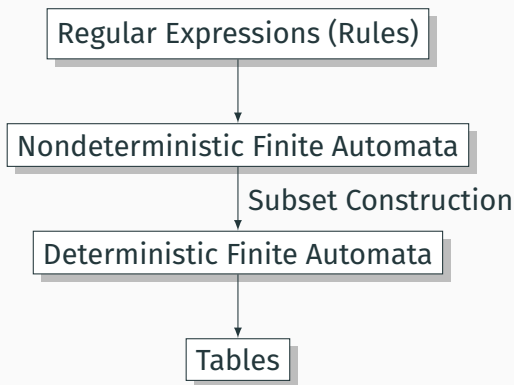
Typical choice:  $\Sigma = \text{ASCII characters, i.e.,}$   
 $\{\_, !, ", \#, \$, \dots, 0, 1, \dots, 9, \dots, A, \dots, Z, \dots, \sim\}$

**letters:**  $A \mid B \mid \dots \mid Z \mid a \mid \dots \mid z$

**digits:**  $0 \mid 1 \mid \dots \mid 9$

**identifier:**  $\text{letter} (\text{letter} \mid \text{digit})^*$

# Implementing Scanners Automatically

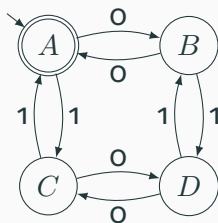


# Nondeterministic Finite Automata

“All strings containing an even number of 0's and 1's”

# Nondeterministic Finite Automata

“All strings containing an even number of 0’s and 1’s”



1. Set of states

$$S : \left\{ \textcircled{\textcircled{A}} \textcircled{B} \textcircled{C} \textcircled{D} \right\}$$

2. Set of input symbols  $\Sigma : \{0, 1\}$

3. Transition function

$$\sigma : S \times \Sigma_{\epsilon} \rightarrow 2^S$$

state	$\epsilon$	0	1
A	$\emptyset$	$\{B\}$	$\{C\}$
B	$\emptyset$	$\{A\}$	$\{D\}$
C	$\emptyset$	$\{D\}$	$\{A\}$
D	$\emptyset$	$\{C\}$	$\{B\}$

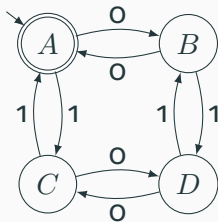
4. Start state  $s_0 : \textcircled{\textcircled{A}}$

5. Set of accepting states

$$F : \left\{ \textcircled{\textcircled{A}} \right\}$$

## The Language induced by an NFA

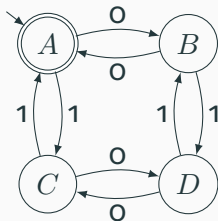
An NFA accepts an input string  $x$  iff there is a path from the start state to an accepting state that “spells out”  $x$ .



Show that the string “010010” is accepted.

# The Language induced by an NFA

An NFA accepts an input string  $x$  iff there is a path from the start state to an accepting state that “spells out”  $x$ .



Show that the string “010010” is accepted.



## Translating REs into NFAs (Thompson's algorithm)

$a$



Symbol

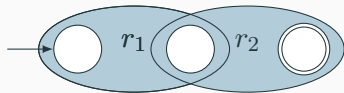
## Translating REs into NFAs (Thompson's algorithm)

$a$



Symbol

$r_1 r_2$



Sequence



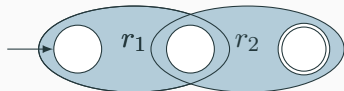
# Translating REs into NFAs (Thompson's algorithm)

$a$



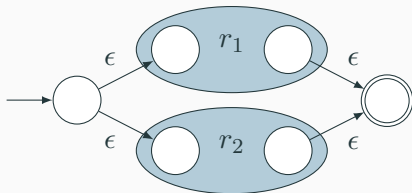
Symbol

$r_1 r_2$



Sequence

$r_1 \mid r_2$



Choice

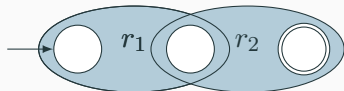
# Translating REs into NFAs (Thompson's algorithm)

$a$



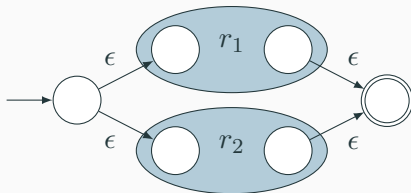
Symbol

$r_1 r_2$



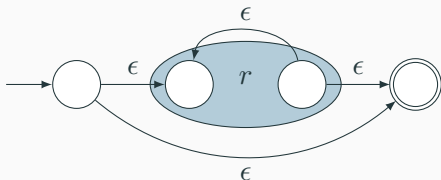
Sequence

$r_1 \mid r_2$



Choice

$(r)^*$

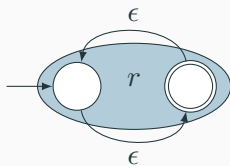


Kleene Closure

## Why So Many Extra States and Transitions?

Invariant: Single start state; single end state; at most two outgoing arcs from any state: helpful for simulation.

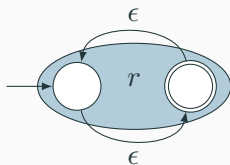
What if we used this simpler rule for Kleene Closure?



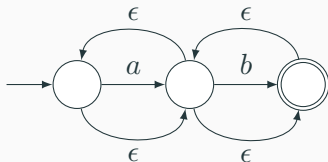
# Why So Many Extra States and Transitions?

Invariant: Single start state; single end state; at most two outgoing arcs from any state: helpful for simulation.

What if we used this simpler rule for Kleene Closure?



Now consider  $a^*b^*$  with this rule:



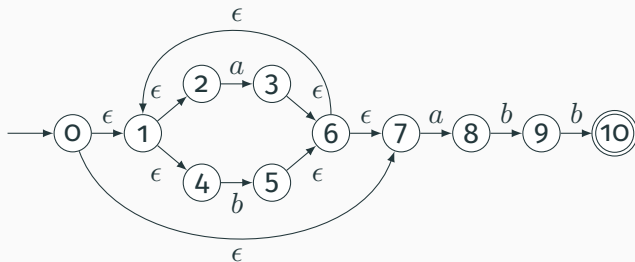
Is this right?

## Translating REs into NFAs

Example: Translate  $(a \mid b)^*abb$  into an NFA. Answer:

## Translating REs into NFAs

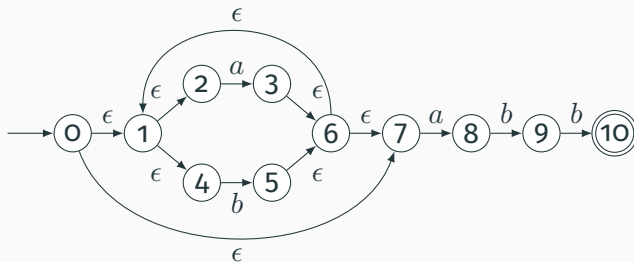
Example: Translate  $(a \mid b)^*abb$  into an NFA. Answer:



Show that the string " $aabb$ " is accepted. Answer:

## Translating REs into NFAs

Example: Translate  $(a \mid b)^*abb$  into an NFA. Answer:



Show that the string “aabb” is accepted. Answer:



## Simulating NFAs

Problem: you must follow the “right” arcs to show that a string is accepted. How do you know which arc is right?



# Simulating NFAs

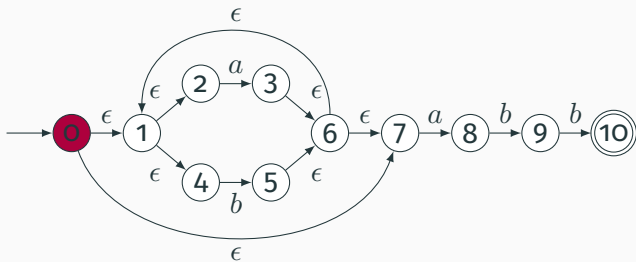
Problem: you must follow the “right” arcs to show that a string is accepted. How do you know which arc is right?

Solution: follow them all and sort it out later.

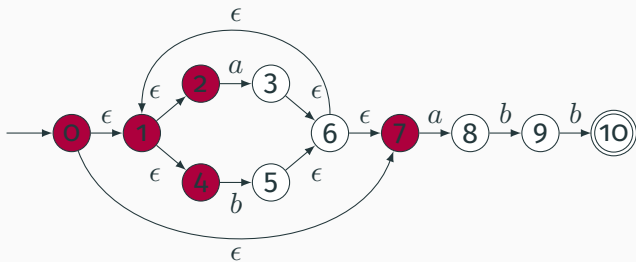
“Two-stack” NFA simulation algorithm:

1. Initial states: the  $\epsilon$ -closure of the start state
2. For each character  $c$ ,
  - New states: follow all transitions labeled  $c$
  - Form the  $\epsilon$ -closure of the current states
3. Accept if any final state is accepting

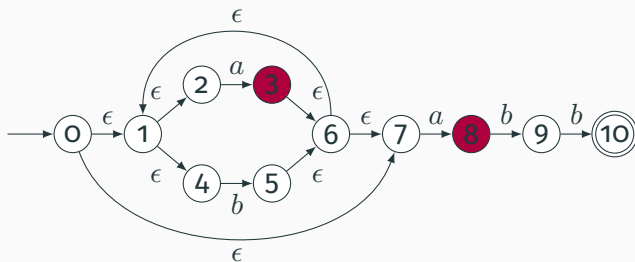
## Simulating an NFA: $\cdot aabb$ , Start



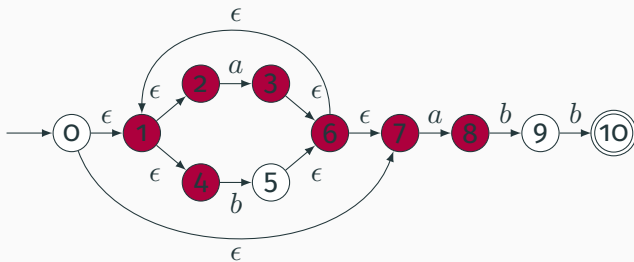
## Simulating an NFA: $\cdot aabb$ , $\epsilon$ -closure



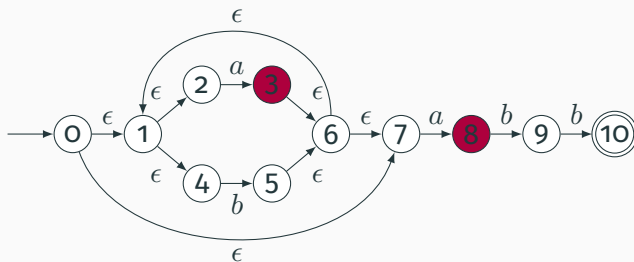
## Simulating an NFA: $a \cdot abb$



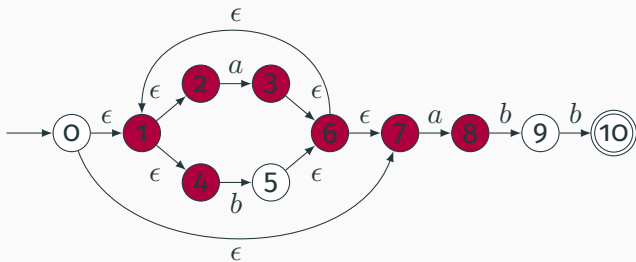
## Simulating an NFA: $a \cdot abb$ , $\epsilon$ -closure



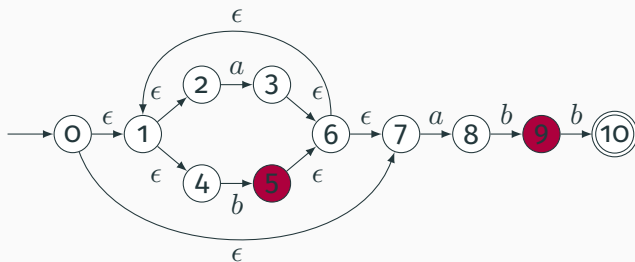
## Simulating an NFA: $aa \cdot bb$



## Simulating an NFA: $aa \cdot bb$ , $\epsilon$ -closure

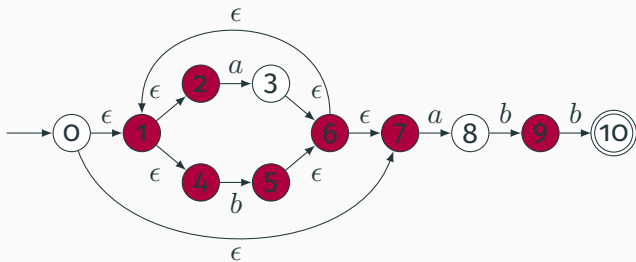


## Simulating an NFA: $aab \cdot b$

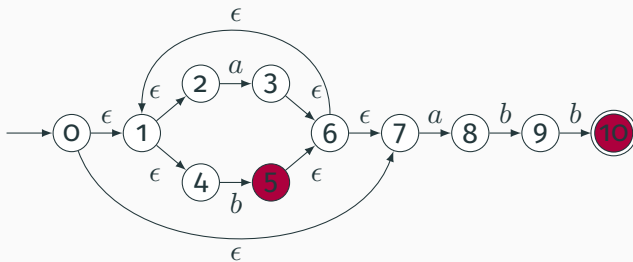




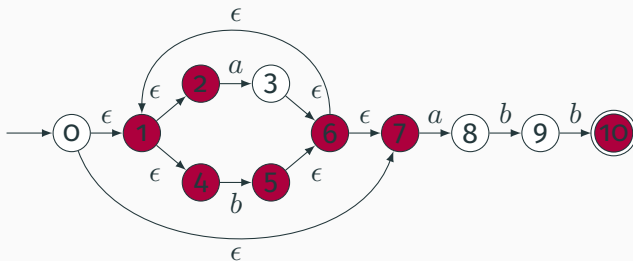
## Simulating an NFA: $aab \cdot b$ , $\epsilon$ -closure



## Simulating an NFA: $aabb$ .



## Simulating an NFA: $aabb$ , Done



# Deterministic Finite Automata

Restricted form of NFAs:

- No state has a transition on  $\epsilon$
- For each state  $s$  and symbol  $a$ , there is at most one edge labeled  $a$  leaving  $s$ .

Very easy to check acceptance: simulate by maintaining current state. Accept if you end up on an accepting state. Reject if you end on a non-accepting state or if there is no transition from the current state for the next symbol.

# Deterministic Finite Automata

```
{  
  type token = ELSE | ELSEIF  
}  
  
rule token =  
  parse "else"   { ELSE }  
    | "elseif" { ELSEIF }
```

# Deterministic Finite Automata

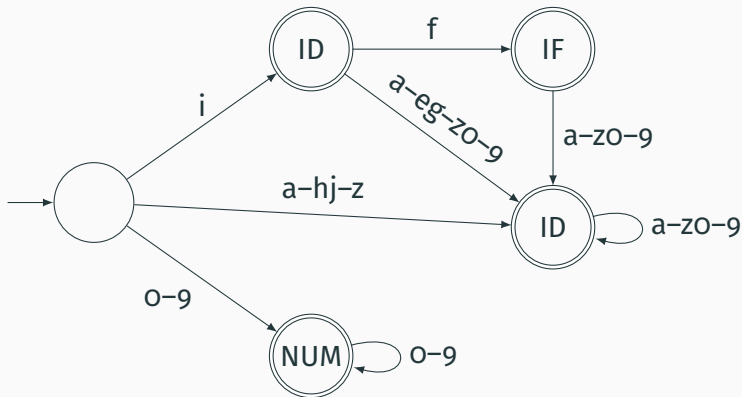
```
{  
  type token = ELSE | ELSEIF  
}  
  
rule token =  
  parse "else"  { ELSE }  
  | "elseif" { ELSEIF }
```



# Deterministic Finite Automata

```
{ type token = IF | ID of string | NUM of string }
```

```
rule token =  
  parse "if" { IF }  
  | ['a'-'z'] ['a'-'z' 'o'-'9']* as lit { ID(lit) }  
  | ['o'-'9']+ as num { NUM(num) }
```



# Building a DFA from an NFA

## Subset construction algorithm

Simulate the NFA for all possible inputs and track the states that appear.

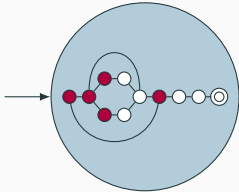
Each unique state during simulation becomes a state in the DFA.



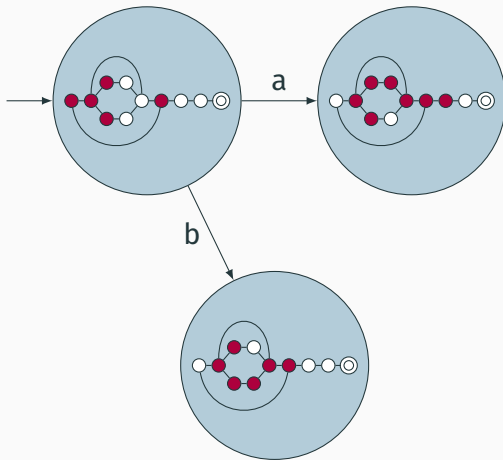
# The Subset Construction Algorithm

1. Create the start state of the DFA by taking the  $\epsilon$ -closure of the start state of the NFA.
2. Perform the following for the new DFA state: For each possible input symbol:
  - Apply move to the newly-created state and the input symbol; this will return a set of states.
  - Apply the  $\epsilon$ -closure to this set of states, possibly resulting in a new set. This set of NFA states will be a single state in the DFA.
3. Each time we generate a new DFA state, we must apply step 2 to it. The process is complete when applying step 2 does not yield any new states.
4. The finish states of the DFA are those which contain any of the finish states of the NFA.

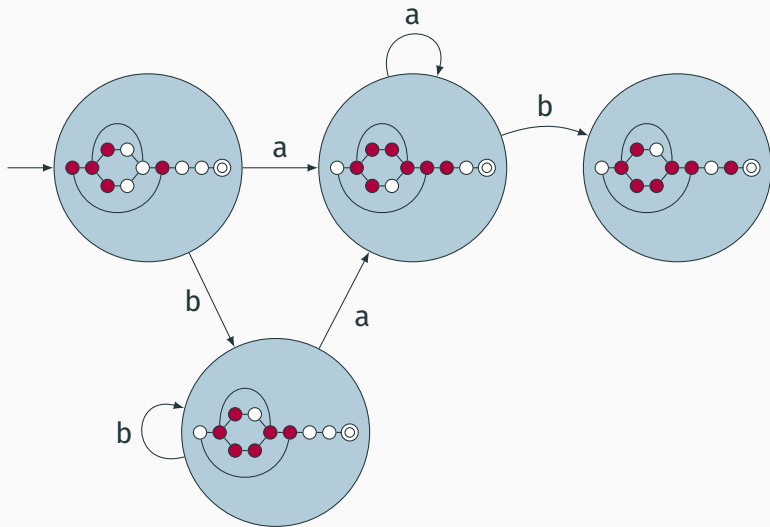
## Subset construction for $(a \mid b)^*abb$



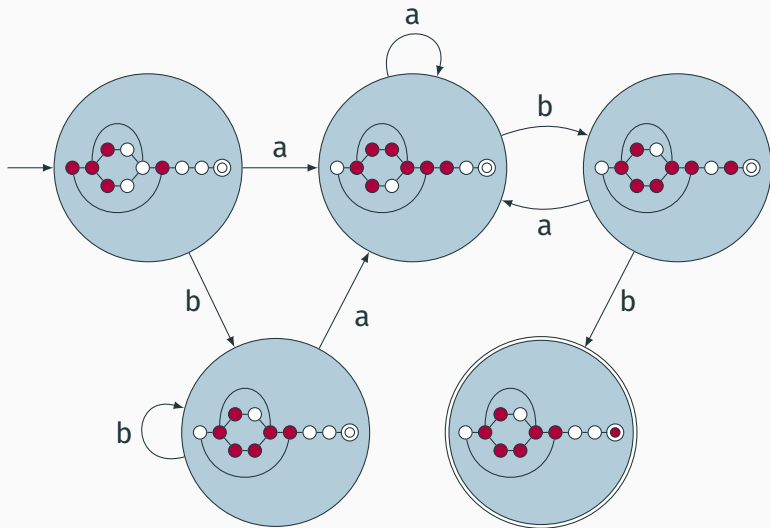
## Subset construction for $(a \mid b)^*abb$



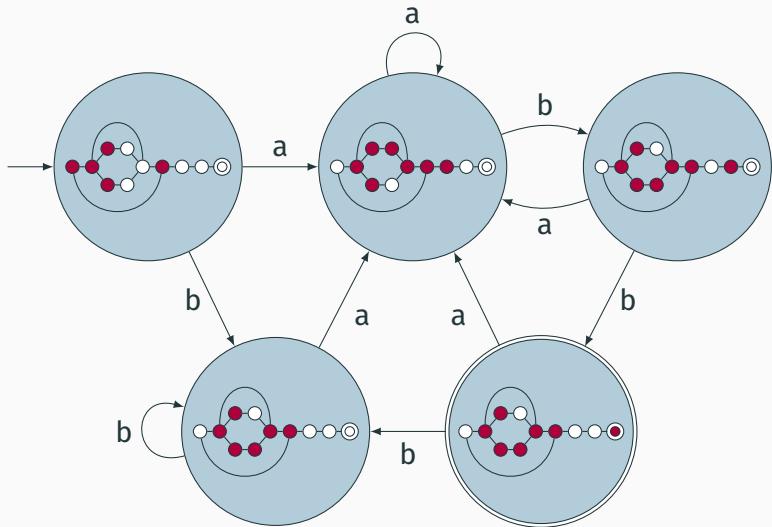
## Subset construction for $(a \mid b)^*abb$



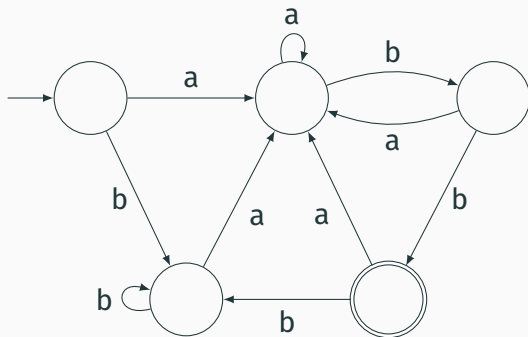
## Subset construction for $(a \mid b)^*abb$



## Subset construction for $(a \mid b)^*abb$

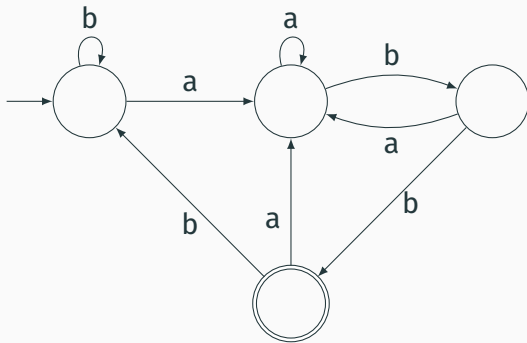


## Result of subset construction for $(a \mid b)^*abb$



*Is this minimal?*

## Minimized result for $(a \mid b)^*abb$

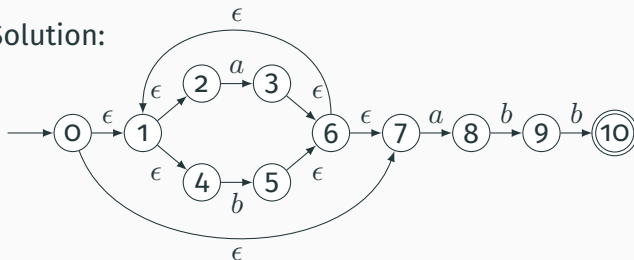




# Transition Table Used In the Dragon Book

Problem: Translate  $(a \mid b)^*abb$  into a DFA.

Solution:

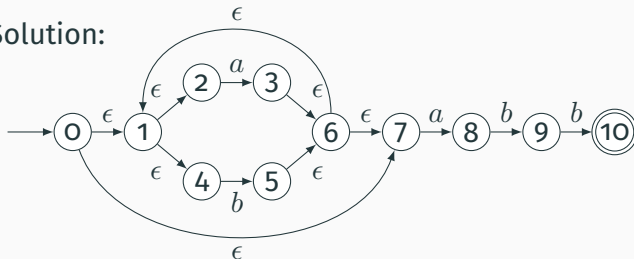


NFA State	DFA State	a	b
-----------	-----------	---	---

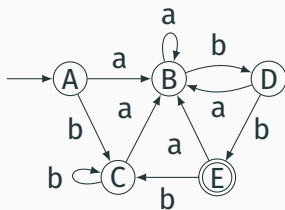
# Transition Table Used In the Dragon Book

Problem: Translate  $(a \mid b)^*abb$  into a DFA.

Solution:



NFA State	DFA State	a	b
$\{0,1,2,4,7\}$	A	B	C
$\{1,2,3,4,6,7,8\}$	B	B	D
$\{1,2,4,5,6,7\}$	C	B	C
$\{1,2,4,5,6,7,9\}$	D	B	E
$\{1,2,4,5,6,7,10\}$	E	B	C



## Subset Construction

An DFA can be exponentially larger than the corresponding NFA.

$n$  states versus  $2^n$

Tools often try to strike a balance between the two representations.

# Lexical Analysis with Ocamllex

---

# Constructing Scanners with Ocamllex



An example:

scanner.mll

```
{ open Parser }

rule token =
  parse [' ' '\t' '\r' '\n'] { token lexbuf }
    | '+' { PLUS }
    | '-' { MINUS }
    | '*' { TIMES }
    | '/' { DIVIDE }
    | ['0'-'9']+ as lit { LITERAL(int_of_string lit) }
    | eof { EOF }
```

# Ocamllex Specifications

```
{  
  (* Header: verbatim OCaml code; mandatory *)  
}  
  
(* Definitions: optional *)  
let ident = regexp  
let ...  
  
(* Rules: mandatory *)  
rule entrypoint1 [arg1 ... argn] =  
  parse pattern1 { action (* OCaml code *) }  
    | ...  
    | patternn { action }  
and entrypoint2 [arg1 ... argn]} =  
  ...  
and ...  
  
{  
  (* Trailer: verbatim OCaml code; optional *)  
}
```

## Patterns (In Order of Decreasing Precedence)

Pattern	Meaning
'c'	A single character
—	Any character (underline)
eof	The end-of-file
"foo"	A literal string
[ '1' '5' 'a'-'z' ]	"1," "5," or any lowercase letter
[ ^ '0'-'9' ]	Any character except a digit
( <i>pattern</i> )	Grouping
<i>identifier</i>	A pattern defined in the <code>let</code> section
<i>pattern</i> *	Zero or more <i>patterns</i>
<i>pattern</i> +	One or more <i>patterns</i>
<i>pattern</i> ?	Zero or one <i>patterns</i>
<i>pattern</i> <sub>1</sub> <i>pattern</i> <sub>2</sub>	<i>pattern</i> <sub>1</sub> followed by <i>pattern</i> <sub>2</sub>
<i>pattern</i> <sub>1</sub>   <i>pattern</i> <sub>2</sub>	Either <i>pattern</i> <sub>1</sub> or <i>pattern</i> <sub>2</sub>
<i>pattern</i> as <i>id</i>	Bind the matched pattern to variable <i>id</i>

# An Example

```
{ type token = PLUS | IF | ID of string | NUM of int }
let letter = ['a'-'z' 'A'-'Z']
let digit = ['0'-'9']

rule token =
  parse [' ' '\n' '\t'] { token lexbuf } (* Ignore whitespace *)

    | '+' { PLUS } (* A symbol *)

    | "if" { IF } (* A keyword *)
    (* Identifiers *)
    | letter (letter | digit | '_' )* as id { ID(id) }
    (* Numeric literals *)
    | digit+ as lit { NUM(int_of_string lit) }

    | "/*" { comment lexbuf } (* C-style comments *)

and comment =
  parse "*/" { token lexbuf } (* Return to normal scanning *)
    | _ { comment lexbuf } (* Ignore other characters *)
```



# Nested Comments

```
{ type token = PLUS | ID of string | NUM of int }

let letter = ['a'-'z' 'A'-'Z']
let digit = ['0'-'9']

rule token =
  parse [' ' '\n' '\t'] { token lexbuf } (* Ignore whitespace *)

    | '+' { PLUS } (* A symbol *)

    | letter (letter | digit | '_' )* as id { ID(id) }
    | digit+ as lit { NUM(int_of_string lit) }

    | "/*" { comment 0 lexbuf } (* C-style comments *)

and comment level =
  parse "*/" { if level == 0 then token lexbuf
               else comments (level - 1) lexbuf }
    | "/*" { comment (level + 1) lexbuf }
    | _ { comment level lexbuf } (* Ignore other characters *)
```

## Free-Format Languages

Typical style arising from scanner/parser division

Program text is a series of tokens possibly separated by whitespace and comments, which are both ignored.

- keywords (if while)
- punctuation (, ( +)
- identifiers (foo bar)
- numbers (10 -3.14159e+32)
- strings ("A String")

# Free-Format Languages

Java      C      C++      C#      Algol      Pascal

Some deviate a little (e.g., C and C++ have a separate preprocessor)

But not all languages are free-format.

The Python scripting language groups with indentation

```
i = 0
while i < 10:
    i = i + 1
    print i      # Prints 1, 2, ..., 10

i = 0
while i < 10:
    i = i + 1
print i          # Just prints 10
```

This is succinct, but can be error-prone.

How do you wrap a conditional around instructions?

## Syntax and Language Design

Does syntax matter? Yes and no

More important is a language's *semantics*—its meaning.

The syntax is aesthetic, but can be a religious issue.

But aesthetics matter to people, and can be critical.

Verbosity does matter: smaller is usually better.

Too small can be problematic: APL is a succinct language with its own character set.

There are no APL programs, only puzzles.

# Syntax and Language Design

Some syntax is error-prone. Classic FORTRAN example:

```
DO 5 I = 1,25 ! Loop header (for i = 1 to 25)  
DO 5 I = 1.25 ! Assignment to variable DO5I
```

Trying too hard to reuse existing syntax in C++:

```
vector< vector<int> > foo;  
vector<vector<int>> foo; // Syntax error
```

C distinguishes `>` and `>>` as different operators.

Bjarne Stroustrup tells me they have finally fixed this.