

Scanning and Parsing

Stephen A. Edwards

Columbia University

Fall 2018



The First Question

How do you represent one of many things?

*Compilers should accept many programs;
how do we describe which one we want?*

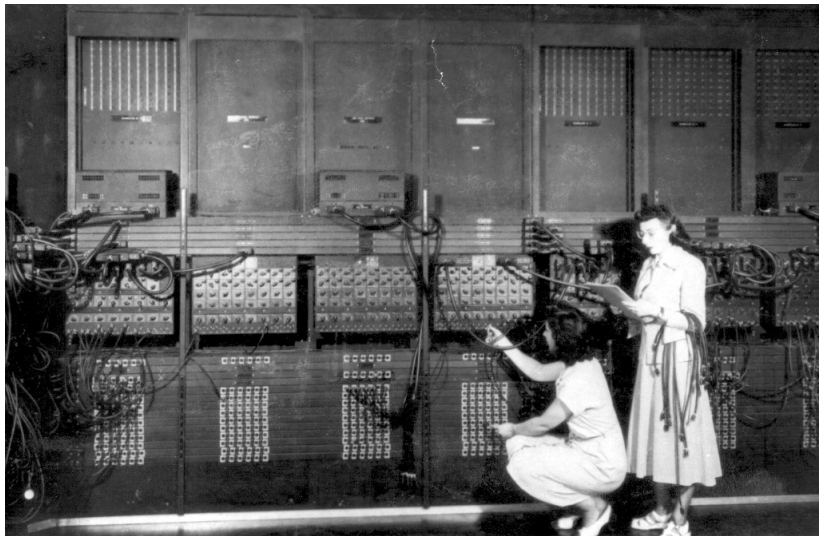
Use continuously varying values?



Very efficient, but has serious noise issues

Edison Model B Home Cylinder phonograph, 1906

The ENIAC: Programming with Spaghetti



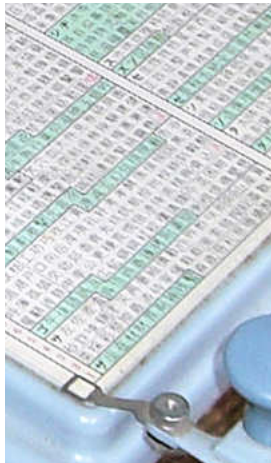
Have one symbol per thing?



Works nicely when there are only a few things

Sholes and Glidden Typewriter, E. Remington and Sons, 1874

Have one symbol per thing?

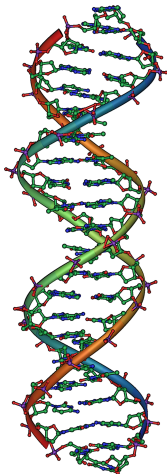


Not so good when there are many, many things

Nippon Typewriter SH-280, 2268 keys

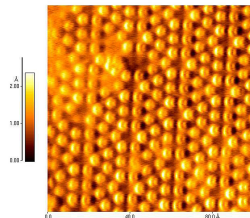
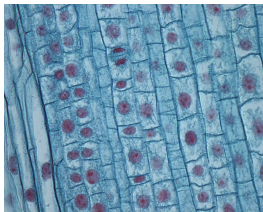
Solution: Use a Discrete Combinatorial System

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



ENGLISH SOUNDS

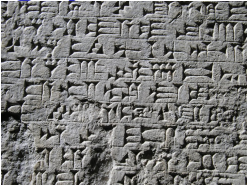
chance	flck	voosle	vooc	roaf	rnslg		
elephant	camero	skrk	bell	clare	lady	pinole	
fat	hut	car	lock	valy	brife	cow	
pot	voable	table	stir	clapper	jeep	key	gnost
power	vun	stunio	seafight	sasle	vage	shower	kick
mouse	no-explur	song	house	light	ring	welt	yacht



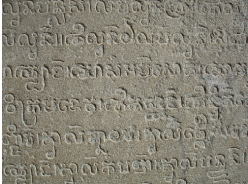
Every Human Writing System Does This



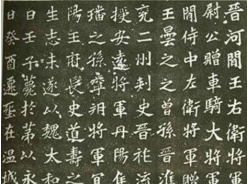
Hieroglyphics (24+)



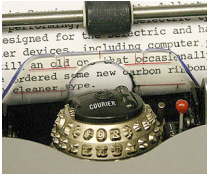
Cuneiform (1000 - 300)



Sanskrit (36)



Chinese (214 - 4000)



IBM Selectric (88-96)



Mayan (100)



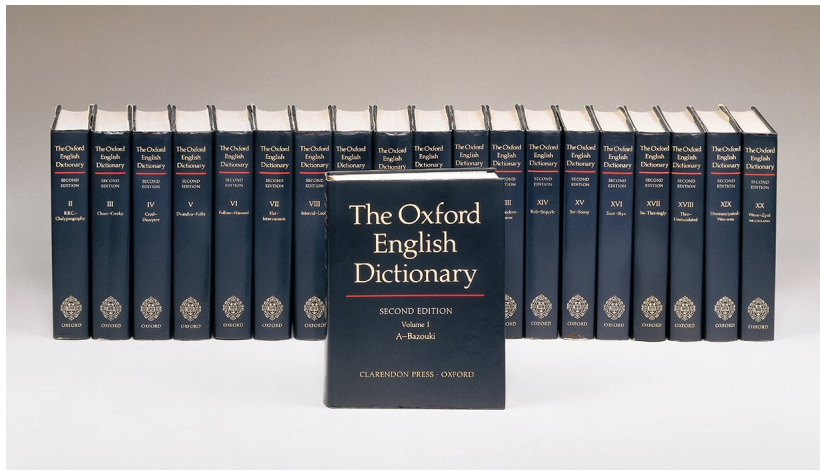
Roman (21-26)

The Second Question

How do you describe only certain combinations?

*Compilers should only accept correct programs;
how should a compiler check that its input is correct?*

Just List Them?



Gets annoying for large numbers of combinations

Just List Them?

3 AA—AAAAA AAAAAA A

AAAAA Budget Moving
16 WilbyCr. 241-5468

AAAAA Canadian Mini-Warehouse
Properties 5399 EgintonW. 620-1577
1001 Arrowld. 742-0228
24 Jeffersonlv. 523-7572
4120 FinchC. 298-3126

AAAAA Criter Control. 201-4711
AAAAA Criter Control. 201-4711
AAAAA 100 Bestest Uninvolve. 410-8727
AAAAA A Devco Glass 410-0371
AAAAA Dreamworks Ltd
Toronto East. 422-0501
AAAAA A Evening Rendezvous. 929-6848
AAAAA EIH Mini Storage
555 ThrethrewyDr. 247-6294
AAAAA A European. 962-2033
AAAAA Expert Movers 16 WilbyCr. 242-7478
AAAAA A Jewel Of The Orient. 929-9975
AAAAA A Limousine Connection.
The 967-5466
AAAAA A Mature Escorts 925-5433
AAAAA A Move Master 588-4656
AAAAA A Neel Professional Moving
Systems 2480 LawrenceAve. 285-6325
AAAAA A Prince Claude Moving. 287-6701
AAAAA Silk Stockings. 534-3509
AAAAA A Woodbine Moving/Storage Ltd
65 Brockford. 751-4900
AAAAA A Alert Glass/Mirror 638-1989
AAAAA A All Star Movers
603 Evans. 259-1578
AAAAA A Armstrong Moving/Storage
Storage. 233-2477
AAAAA A HSL Moving/Storage
603 Evans. 259-7250
AAAAA A Midrup Moving/Storage
50 EmeraldCr. 494-9451
AAAAA A-1 Moving/Storage
637 Lansdowne. 516-3536
AAAAA A Prestige Movers
703 GladstoneAve. 533-2633
AAAAA A Southern Ontario Wildlife
Removal. 690-4066
AAAAA A Speedy Moving
124 Crockford. 285-0084
A-A-A-A-A Speedy Moving
1540 VictoriaPark. 751-9532
AAAAA A A Across The World Courier
425 AdelaideW. 504-0008
AAAAA A Auto Glass
855 Alhess. 663-8676
AAAAA A California Dreams Escort
Service. 323-3899
AAAAA A California Dreams Massage
Service. 323-3899
AAAAA A National Auto Glass
Escorts. 622-1177
AAAAA A A A A A Best Of The
562 Kipling. 503-3833
AAAAA A A NightDay 929-8975
AAAAA A Strip 'N Tell 964-7877
AAAAA A A Unforgettable Escorts 398-5337
AAAAA A A Automated Door
Systems 22 Juliard. 255-7127
AAAAA A A California Beach Club Escort
Service. 323-9822

AAAAAAA CBS Moving
130 Lansdowne. 533-7139
AAAAAAA Dream Girls 255-5032
AAAAAAA A Big Apple Escort
Service. 465-2767
AAAAAAA Accident And
Accompanying Injuries/Criminal
Practice 1018 FinchW. 663-2211
AAAAAAA A A Accident
Accompanying Injuries/Criminal
Practice 1018 FinchW. 663-2211
AAAAAAA China Blue Escort
Service. 323-9522
AAAAAAA A A A A A ABCDO Door Co
1860 BonhillRd Mississauga. 748-3667
AAAAAAA A A A A A Action
5233 DundasStW. 253-0888
AAAAAAA A A A A A Alert Auto
Glass. 398-4585
AAAAAAA A A A A A A
Or 599-3410
AAAAAAA A A A A A AJMI Campbell Van Lines
Inc. 1190 MyersStDr. 213-5660
AAAAAAA A A A A A A Auto Glass
Hotline. 283-0042
AAAAAAA A A A Collins & Greig
Cartage Ltd 33 Coronet. 239-2991
AAAAAAA A A A Competition Auto
Glass. 223-1292
AAAAAAA A A A Competition Auto
Glass. 283-0042
AAAAAAA A A A Competition Auto
Glass. 410-7693
AAAAAAA A A A International
Escorts. 299-6848
AAAAAAA A A A Jewel Dating/Escort
Service. 461-0629
AAAAAAA A A A Marketing
Services. 413-0444
AAAAAAA A A A Nothing But
Class. 595-1884
AAAAAAA A A A On The Wild Side
Sensational Female Escort Service. 255-1320
AAAAAAA A A A The Good Life Clubs
21 McCaul. 979-1422
AAAAAAA A A A A A A 667-0574
AAAAAAA A A A A A A 297-7279
AAAAAAA A A A A A A 239-2783
AAAAAAA A A A A A A 667-0470
AAAAAAA A A A A A A Affordable And
Aggressive Defence 4950 YongeSt. 221-7108
AAAAAAA A A A A A A Campbell Moving
Services. 285-4433
AAAAAAA A A A A A-1 Windshields To
Go 159 Dymover. 787-8039
AAAAAAA A A A A A Sunrest
Escorts. 622-1177
AAAAAAA A A A A A A Best Of The
Best. 929-3039
AAAAAAA A A A A A A Bill & Son Towing
286 RoyalYork. 253-8518

AAAAAAA AAAAA A

AAAAAAA A A A A A Class Above
Limousine 173 DanforthAve. 465-5643
AAAAAAA A A A A A Cross Movers
1232-B Woodbine. 423-0239
AAAAAAA A A A A A A Miss
Victoria. 967-7176
AAAAAAA A A A A A A Payless And
Escorts. 485-5333
AAAAAAA A A A A A A A A A
AAAAAAA 700 Lawrence/Win. 256-1600
AAAAAAA A A A A A A A A A A A A
699-6700
AAAAAAA A A A A A A A A A A A A
Mannie Zeller 255 DuncanMillRd. 441-9500
AAAAAAA A A A A A A A A A A A A
Associates 1 StClairE. 233-0907
AAAAAAA A A A A A A A A A A A A
Transmissions 285 OldKington. 287-0000
AAAAAAA A A A A A A A A A A A A
Movers/Storage 17 Canso. 242-6662
AAAAAAA A A A A A A A A A A A A A
Storage. 366-0237
AAAAAAA A A A A A A A A A A A A
14-A Hazleton. 964-0138
AAAAAAA A A A A A A A A A A A A
Mover 64 StClairW. 944-2018
AAAAAAA A A A A A A A A A A A A
Abba Auto Collision—
Glass. 777-9595
AAAAAAA A A A A A A A A A A A A
Armor Lock And
Safe 6083 Yonge. 225-5589
AAAAAAA A A A A A A A A A A A A
Basement Systems
Canada 38 Gairforth. 285-6002
AAAAAAA A A A A A A A A A A A A
Law 222-7889
AAAAAAA A A A A A A A A A A A A
If Busy Call 222-7589
AAAAAAA A A A A A A A A A A A A
33 IsabelL. 222-6311
AAAAAAA A A A A A A A A A A A A
55 TownCentre Court
222-5867
AAAAAAA A A A A A A A A A A A A
Ad 3420 FinchE. 499-2144
AAAAAAA A A A A A A A A A A A A
Law 395 Millar. 299-6688
AAAAAAA A A A A A A A A A A A A A
Action Law 5233 DundasStW. 253-0888
AAAAAAA A A A A A A A A A A A A A
AllanAssociates 401 Bay. 363-5431
AAAAAAA A A A A A A A A A A A A A
W Auto Glass 821 Kipling. 233-4773
AAAAAAA A A A A A A A A A A A A A
Alarms 557 DixonRd. 247-0000
AAAAAAA A A A A A A A A A A A A
Towing 10 Canso. 245-7676

AAAAAAA AAAAA A

AAAAAAA AAAAA A A A A A A A A A
Towing 10 Canso. 245-7676
AAAAAAA A A A A A A A A A A A A A
Robertson Moving/Storage
236 NorthQue. 620-1212
AAAAAAA A A A A A A A A A A A A A
Rezz. 652-5252
AAAAAAA A A A A A A A A A A A A A
Access Law 784-2020
AAAAAAA A A A A A A A A A A A A A
Accident
Accompanying Injuries/Criminal
Practice 1000 FinchW. 663-2211
AAAAAAA A A A A A A A A A A A A A
Clubs 2 StClairW. 964-2313
AAAAAAA A A A A A A A A A A A A A
Ability 2 SheppardAve. 224-5750
AAAAAAA A A A A A A A A A A A A A
A A A A A A A A A A A A A A A A A A
Advant Edge Door Systems. 222-0322
AAAAAAA A A A A A A A A A A A A A
Executive's Choice. 929-9390
AAAAAAA A A A A A A A A A A A A A
Automatic Garage Doors
64 Clarkson. 785-7820
252-5686
AAAAAAA A A A A A A A A A A A A A
Alarms 280 Consumers. 494-9777
AAAAAAA A A A A A A A A A A A A A
Bigant
Mature Escorts. 923-3333
AAAAAAA A A A A A A A A A A A A A
Professional Express System
425 AdelaideW. 504-9111
AAAAAAA A A A A A A A A A A A A A
Escorts/Buy. 259-3940
AAAA AAAA AAAA AAAA Anthony De
Marco 1205 StClairE. 651-2299
AAAA AAAA AAAA AAAA Dominic
Tagliola 1205 StClairW. 651-2299
AAAAAAA A A A A A A A A A A A A A
Always Available. 463-9191
AAAAAAA A A A A A A A A A A A A A
A Touch Of
Class Escort Service. 461-8110
AAAAAAA A A A A A A A A A A A A A
Apple Auto Glass
No Charge-Dial 1 800 506-5665
AAAAAAA A A A A A A A A A A A A A
Cardinal Custom Building 2 MoorW. 956-4728
AAAA A A A A A A A A A A A A A A A A
LU Storing Movers. 693-2403
A A A A A A A A A A A A A A A A A A
1860 BonhillRd Mississauga
Toronto 748-3667
AAAAA A B S Movers
643 LansdowneAve. 588-1499
AA A A A B C D E F Locksmith
80 StClairE. 922-2255
AAAAA A B C Movers Inc.
Columbus. 535-3413
AAAAA A B S Best Movers 503-9321
AAAAA M O 1 Moving Systems
195 Middlefield. 299-4223
AAAAA A A A B B Moving 900
Caledonia. 787-4654
A A A A B B B E Locksmiths 287-6001
A A A A B C Glass Supply 11 Concord. 531-1548
A A A A B C D E F Window Co
1860 BonhillRd Mississauga
Toronto 748-3667

Can be really redundant

Choices: CS Research Jargon Generator

Pick one from each column

an integrated

a parallel

a virtual

an interactive

a responsive

a synchronized

a balanced

a virtual

a meta-level

mobile

functional

programmable

distributed

logical

digital

concurrent

knowledge-based

multimedia

network

preprocessor

compiler

system

interface

protocol

architecture

database

algorithm

E.g., "a responsive knowledge-based preprocessor."

<http://www.cs.purdue.edu/homes/dec/essay.topic.generator.html>

Router: A Methodology for the Typical Unification of Access Points and Redundancy

Jeremy Stribling, Daniel Aguayo and Maxwell Krohn

ABSTRACT

Many physicists would agree that, had it not been for congestion control, the evaluation of web browsers might never have occurred. In fact, few hackers worldwide would disagree with the essential unification of voice-over-IP and public-private key pair. In order to solve this riddle, we confirm that SMPs can be made stochastic, cacheable, and interoperable.

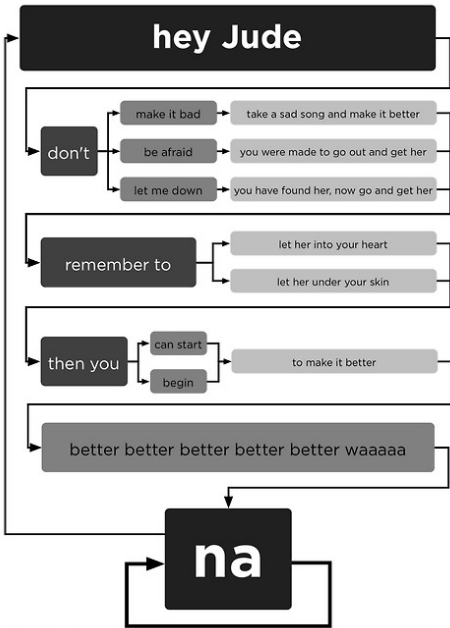
I. INTRODUCTION

Many scholars would agree that, had it not been for active networks, the simulation of Lamport clocks might never have occurred. The notion that end-users synchronize with the investigation of Markov models is rarely outdated. A theoretical grand challenge in theory is the important unification

The rest of this paper is organized as follows: we motivate the need for fiber-optic cable; we work in context with the prior work in the field; to address this obstacle, we disprove that even the most touted autonomous algorithm for the construction of analog converters by Jones [10] is NP-complete; oriented languages can be made signed, deterministic, and signed. Along these same lines, to accomplish our goal, we concentrate our efforts on showing that the fastest algorithm for the exploration of robots by Stribling is $\Omega((n + \log n))$ time [22]. In the end, we conclude.

II. ARCHITECTURE

Our research is principled. Consider the example of the work by Martin and Smith; our model is similar, but more complex.



How about more structured collections of things?

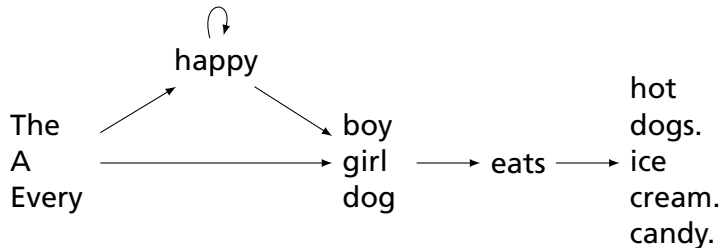
The boy eats hot dogs.

The dog eats ice cream.

Every happy girl eats candy.

A dog eats candy.

The happy happy dog eats hot dogs.



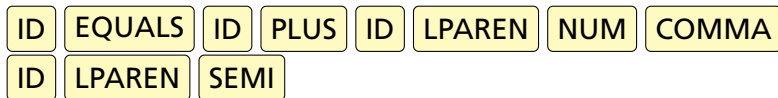
Lexical Analysis

Lexical Analysis (Scanning)

Translate a stream of characters to a stream of tokens



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



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[†]

Scanners are usually much faster than parsers.

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

Parser does not care that the the identifier is "supercalifragilisticexpialidocious."

Parser rules are only concerned with tokens.

[†] 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

String: A finite sequence of symbols from an alphabet

Examples: ϵ (the empty string), Stephen, $\alpha\beta\gamma$

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

Concatenation: Strings from one followed by the other

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

Union: All strings from each language

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

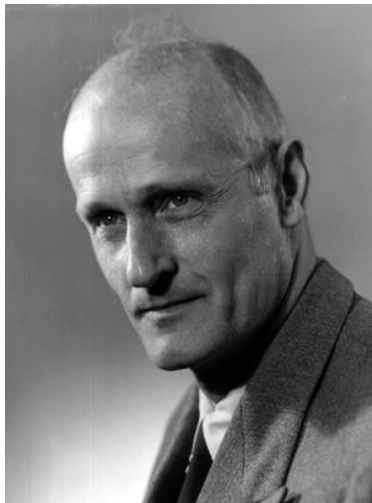
Kleene Closure: Zero or more concatenations

$M^* = \{ \epsilon \} \cup M \cup MM \cup MMM \dots =$
 $\{ \epsilon, man, men, manman, manmen, menman, menmen,$
 $manmanman, manmanmen, manmenman, \dots \}$

Kleene Closure

"*" is named after Stephen Cole Kleene, the inventor of regular expressions, who pronounced his last name "CLAY-nee."

His son Ken writes "As far as I am aware this pronunciation is incorrect in all known languages. I believe that this novel pronunciation was invented by my father."



Regular Expressions over an Alphabet Σ

A standard way to express languages for tokens.

1. ϵ 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 $L(s)$,

$(r) | (s)$ 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
$a \mid b$	$\{a, b\}$
$(a \mid b)(a \mid b)$	$\{aa, ab, ba, bb\}$
a^*	$\{\epsilon, a, aa, aaa, aaaa, \dots\}$
$(a \mid b)^*$	$\{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, \dots\}$
$a \mid a^*b$	$\{a, b, ab, aab, aaab, \dots\}$

Specifying Tokens with REs

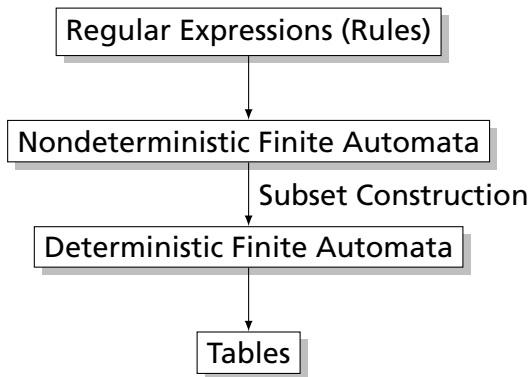
Typical choice: $\Sigma = \text{ASCII characters, i.e.,}$
 $\{_, !, ", \#, \$, \dots, 0, 1, \dots, 9, \dots, A, \dots, Z, \dots, \sim\}$

letters: $A | B | \dots | Z | a | \dots | z$

digits: $0 | 1 | \dots | 9$

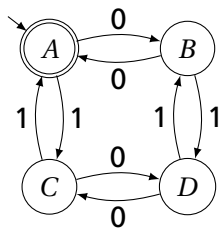
identifier: $\text{letter}(\text{letter} | \text{digit})^*$

Implementing Scanners Automatically



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 $\sigma: S \times \Sigma_c \rightarrow 2^S$

state	ϵ	0	1
A	\emptyset	{B}	{C}
B	\emptyset	{A}	{D}
C	\emptyset	{D}	{A}
D	\emptyset	{C}	{B}

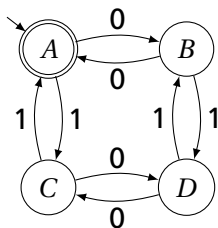
4. Start state $s_0: \textcircled{\textcircled{A}}$

5. Set of accepting states

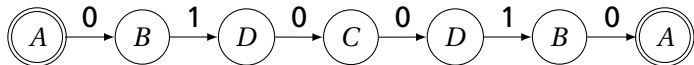
$$F: \left\{ \textcircled{\textcircled{A}} \right\}$$

The Language induced by an NFA

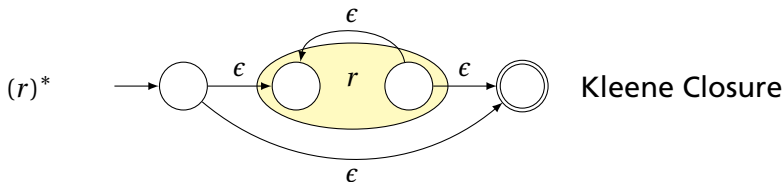
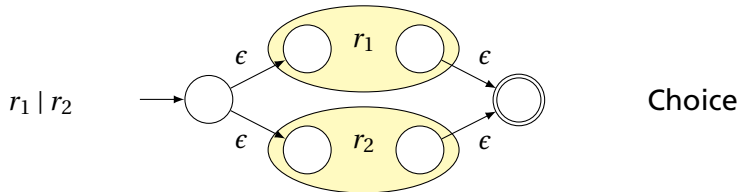
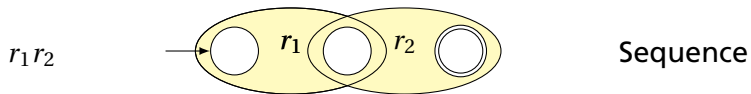
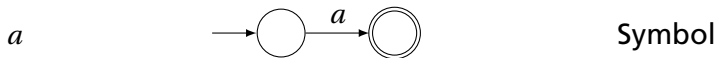
An NFA accepts an input string x iff there is a path from the start state to an accepting state that "spells out" x .



Show that the string "010010" is accepted.



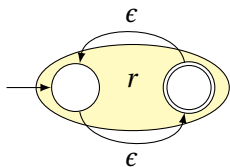
Translating REs into NFAs (Thompson's algorithm)



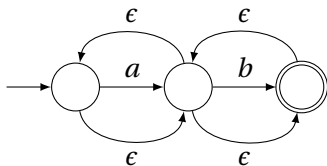
Why So Many Extra States and Transitions?

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?



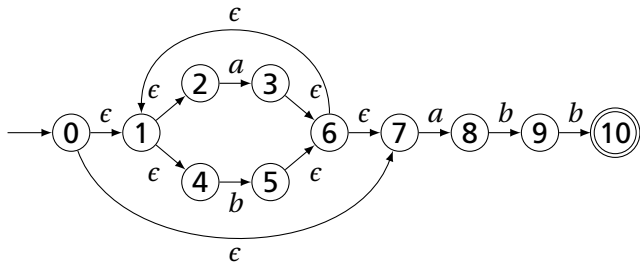
Now consider a^*b^* with this rule:



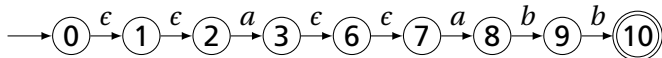
Is this right?

Translating REs into NFAs

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



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



Simulating NFAs

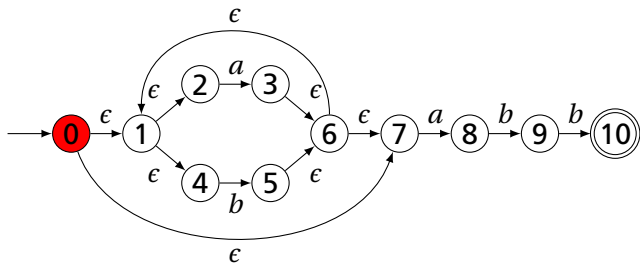
Problem: you must follow the “right” arcs to show that a string is accepted. How do you know which arc is right?

Solution: follow them all and sort it out later.

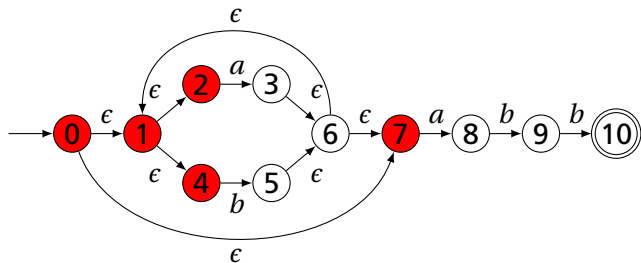
“Two-stack” NFA simulation algorithm:

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

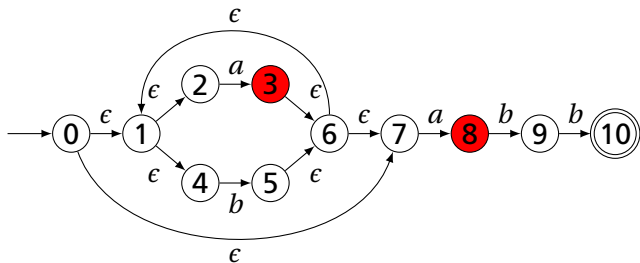
Simulating an NFA: $\cdot aabb$, Start



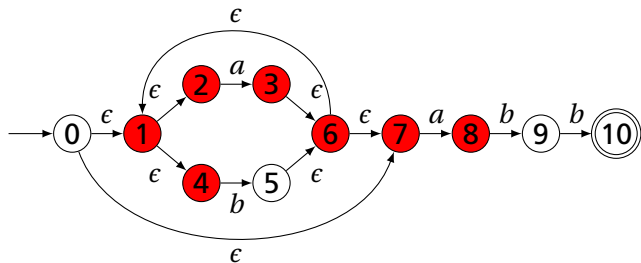
Simulating an NFA: $\cdot aabb$, ϵ -closure



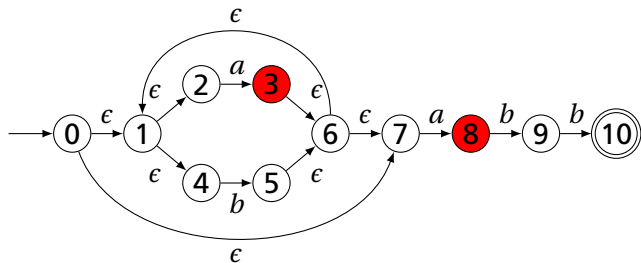
Simulating an NFA: $a \cdot abb$



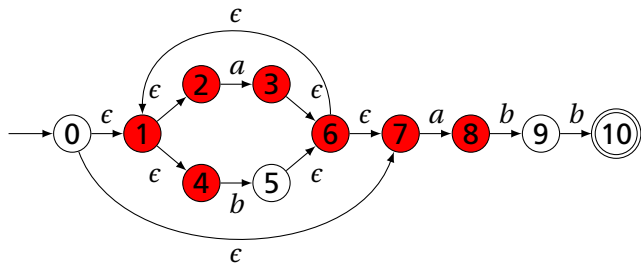
Simulating an NFA: $a \cdot abb$, ϵ -closure



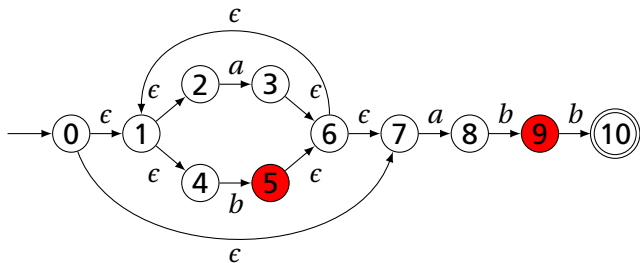
Simulating an NFA: $aa \cdot bb$



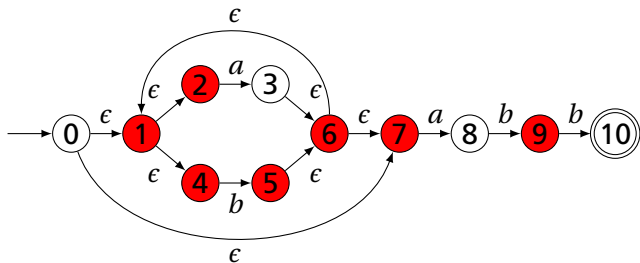
Simulating an NFA: $aa \cdot bb$, ϵ -closure



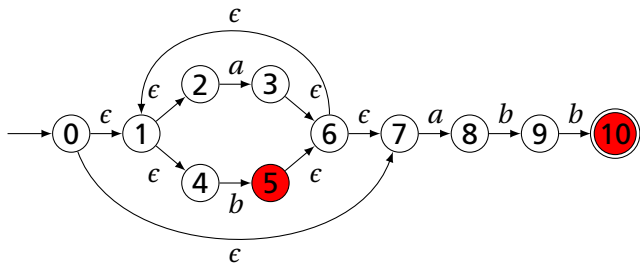
Simulating an NFA: $aab \cdot b$



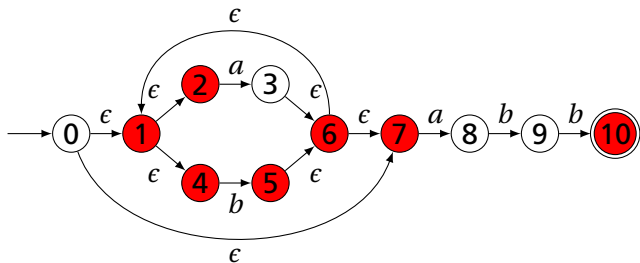
Simulating an NFA: $aab \cdot b$, ϵ -closure



Simulating an NFA: $aabb$.



Simulating an NFA: $aabb\cdot$, Done



Deterministic Finite Automata

Restricted form of NFAs:

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

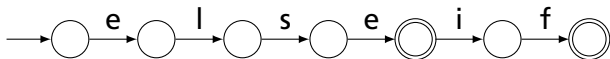
Differs subtly from the definition used in COMS W3261
(Sipser, *Introduction to the Theory of Computation*)

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

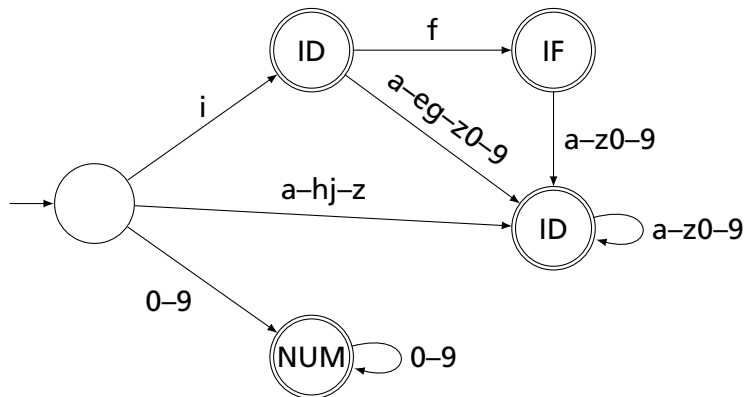
```
{ type token = IF | ID of string | NUM of string }
```

```
rule token =
```

```
  parse "if"
```

```
    | ['a'-'z'] ['a'-'z' '0'-'9']* as lit { ID(lit) }
```

```
    | ['0'-'9']+ as num { NUM(num) }
```



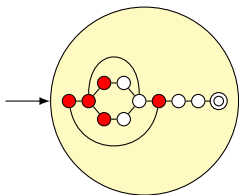
Building a DFA from an NFA

Subset construction algorithm

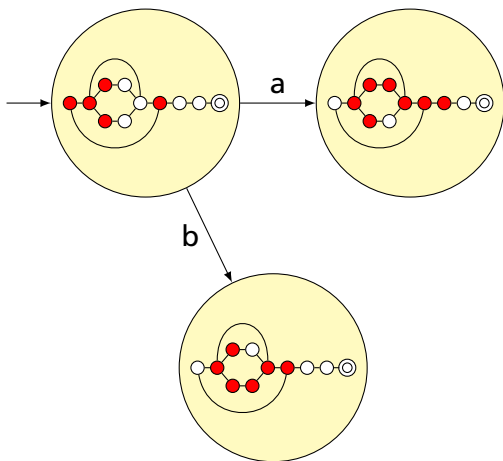
Simulate the NFA for all possible inputs and track the states that appear.

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

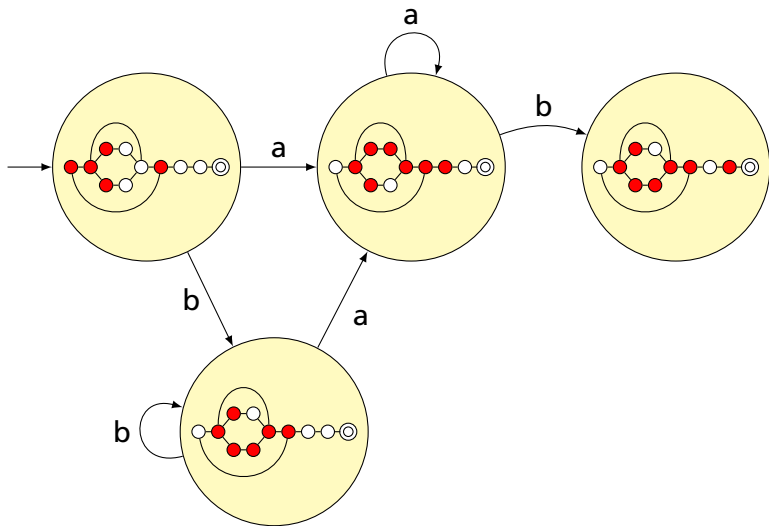
Subset construction for $(a | b)^* abb$



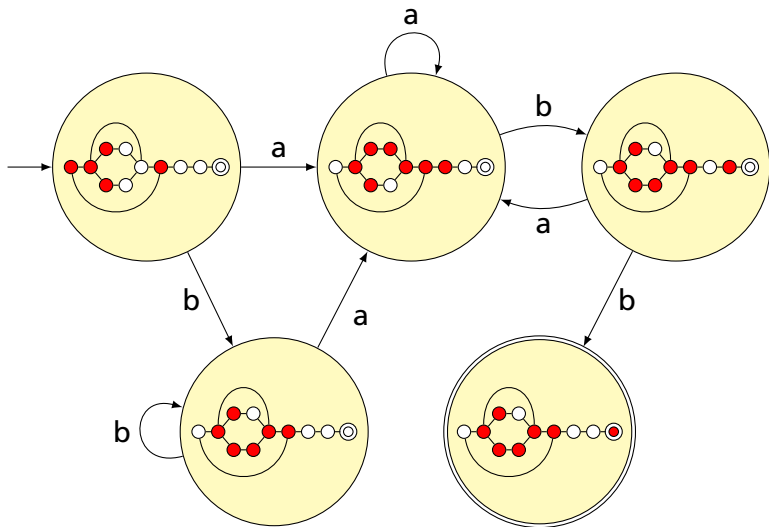
Subset construction for $(a|b)^*abb$



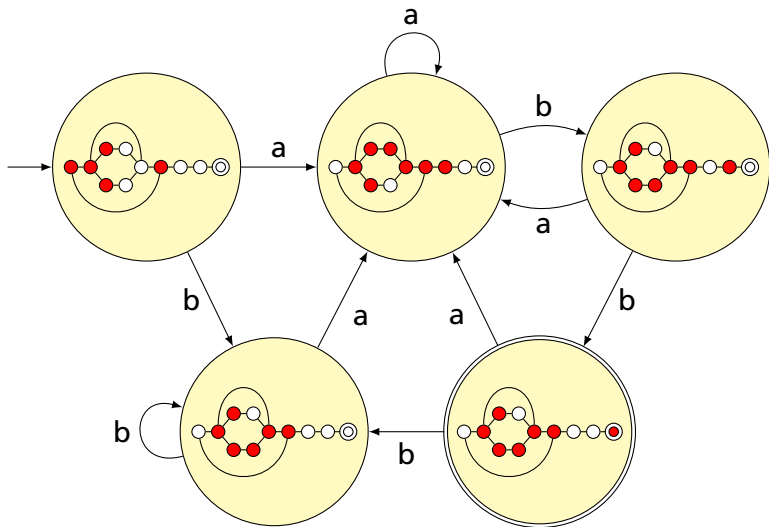
Subset construction for $(a|b)^*abb$



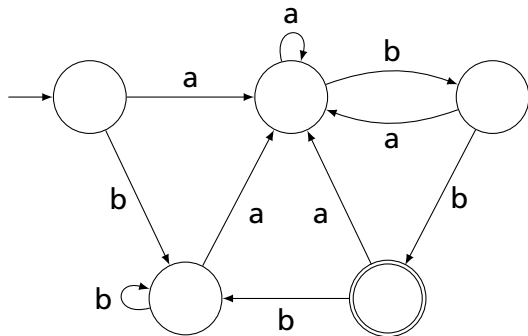
Subset construction for $(a|b)^*abb$



Subset construction for $(a|b)^*abb$

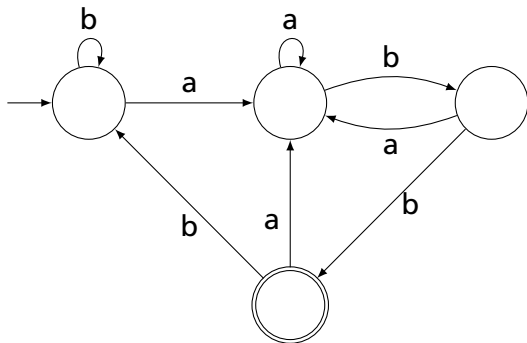


Result of subset construction for $(a|b)^*abb$



Is this minimal?

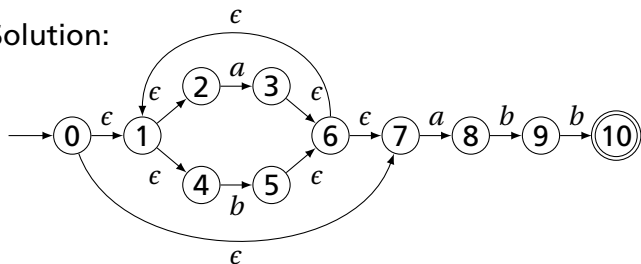
Minimized result for $(a | b)^* abb$



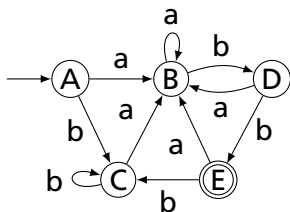
Transition Table Used In the Dragon Book

Problem: Translate $(a | b)^* abb$ into an NFA and perform subset construction to produce 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

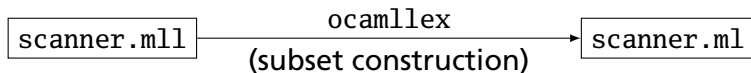
An DFA can be exponentially larger than the corresponding NFA.

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 }
```


Ocamlex 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 1et 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> ₁ <i>pattern</i> ₂	<i>pattern</i> ₁ followed by <i>pattern</i> ₂
<i>pattern</i> ₁ <i>pattern</i> ₂	Either <i>pattern</i> ₁ or <i>pattern</i> ₂
<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 *)
  | letter (letter | digit | '_')* as id { ID(id) } (* Identifiers *)
  | digit+ as lit { NUM(int_of_string lit) } (* Numeric literals *)

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

and comment =
  parse "*/" { token lexbuf } (* Return to normal scanning *)
  | _ { comment 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.

FORTRAN 77

FORTRAN 77 is not free-format. 72-character lines:

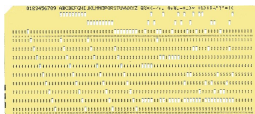
```
100  IF(IN .EQ. 'Y' .OR. IN .EQ. 'y' .OR.  
$    IN .EQ. 'T' .OR. IN .EQ. 't') THEN
```



When column 6 is not a space, line is considered part of the previous.

Fixed-length line works well with a one-line buffer.

Makes sense on punch cards.



Python

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 D05I
```

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.

Modeling Sentences

Simple Sentences Are Easy to Model

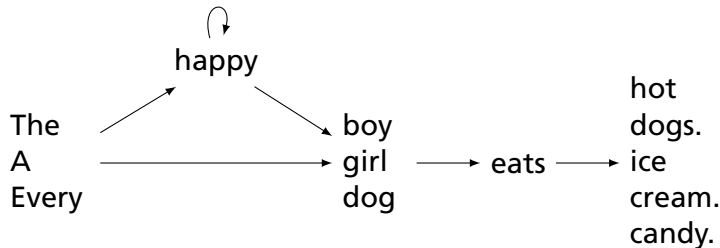
The boy eats hot dogs.

The dog eats ice cream.

Every happy girl eats candy.

A dog eats candy.

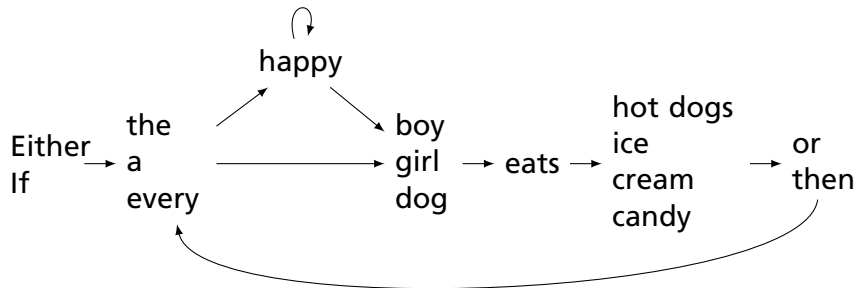
The happy happy dog eats hot dogs.



Richer Sentences Are Harder

If the boy eats hot dogs, then the girl eats ice cream.

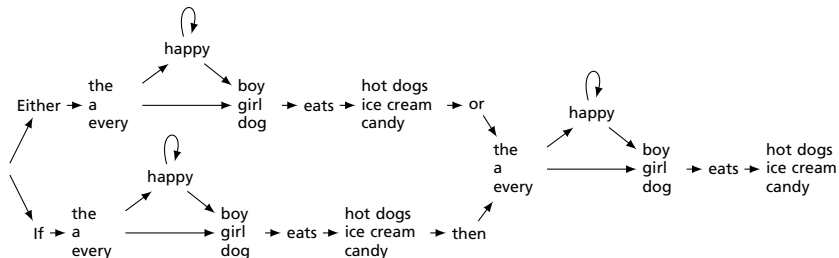
Either the boy eats candy, or every dog eats candy.



Does this work?

Automata Have Poor Memories

Want to “remember” whether it is an “either-or” or “if-then” sentence. Only solution: duplicate states.



Automata in the form of Production Rules

Problem: automata do not remember where they've been

$S \rightarrow \text{Either } A$

$S \rightarrow \text{If } A$

$A \rightarrow \text{the } B$

$A \rightarrow \text{the } C$

$A \rightarrow \text{a } B$

$A \rightarrow \text{a } C$

$A \rightarrow \text{every } B$

$A \rightarrow \text{every } C$

$B \rightarrow \text{happy } B$

$B \rightarrow \text{happy } C$

$C \rightarrow \text{boy } D$

$C \rightarrow \text{girl } D$

$C \rightarrow \text{dog } D$

$D \rightarrow \text{eats } E$

$E \rightarrow \text{hot dogs } F$

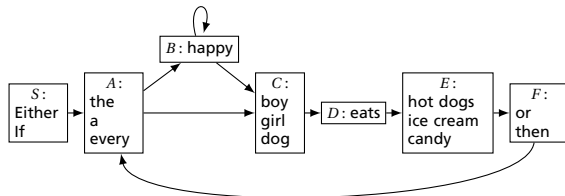
$E \rightarrow \text{ice cream } F$

$E \rightarrow \text{candy } F$

$F \rightarrow \text{or } A$

$F \rightarrow \text{then } A$

$F \rightarrow \epsilon$



Solution: Context-Free Grammars

Context-Free Grammars have the ability to “call subroutines:”

$S \rightarrow$ Either P , or P . Exactly two P s

$S \rightarrow$ If P , then P .

$P \rightarrow A H N$ eats O One each of A , H , N , and O

$A \rightarrow$ the

$A \rightarrow$ a

$A \rightarrow$ every

$H \rightarrow$ happy H H is “happy” zero or more times

$H \rightarrow \epsilon$

$N \rightarrow$ boy

$N \rightarrow$ girl

$N \rightarrow$ dog

$O \rightarrow$ hot dogs

$O \rightarrow$ ice cream

$O \rightarrow$ candy

A Context-Free Grammar for a Simplified C

program $\rightarrow \epsilon \mid \text{program vdecl} \mid \text{program fdecl}$

fdecl $\rightarrow \text{id (formals) \{ vdecls stmts \}}$

formals $\rightarrow \text{id} \mid \text{formals , id}$

vdecls $\rightarrow \text{vdecl} \mid \text{vdecls vdecl}$

vdecl $\rightarrow \text{int id ;}$

stmts $\rightarrow \epsilon \mid \text{stmts stmt}$

stmt $\rightarrow \text{expr ;} \mid \text{return expr ;} \mid \{ \text{stmts} \} \mid \text{if (expr) stmt} \mid$
 $\text{if (expr) stmt else stmt} \mid$
 $\text{for (expr ; expr ; expr) stmt} \mid \text{while (expr) stmt}$

expr $\rightarrow \text{lit} \mid \text{id} \mid \text{id (actuals)} \mid (\text{expr}) \mid$
 $\text{expr + expr} \mid \text{expr - expr} \mid \text{expr * expr} \mid \text{expr / expr} \mid$
 $\text{expr == expr} \mid \text{expr != expr} \mid \text{expr < expr} \mid \text{expr <= expr} \mid$
 $\text{expr > expr} \mid \text{expr >= expr} \mid \text{expr = expr}$

actuals $\rightarrow \text{expr} \mid \text{actuals , expr}$

Constructing Grammars and Ocamlyacc

Parsing

Objective: build an abstract syntax tree (AST) for the token sequence from the scanner.



Goal: verify the syntax of the program, discard irrelevant information, and “understand” the structure of the program.

Parentheses and most other forms of punctuation removed.

Ambiguity

One morning I shot an elephant in my pajamas.

Ambiguity

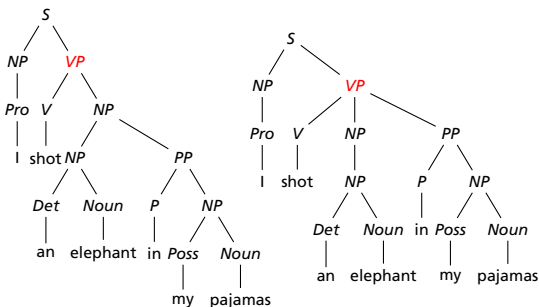
*One morning I shot an elephant in my pajamas.
How he got in my pajamas I don't know. —Groucho Marx*



Ambiguity in English

I shot an elephant in my pajamas

S	→	NP VP
VP	→	V NP
VP	→	V NP PP
NP	→	NP PP
NP	→	Pro
NP	→	Det Noun
NP	→	Poss Noun
PP	→	P NP
V	→	shot
Noun	→	elephant
Noun	→	pajamas
Pro	→	I
Det	→	an
P	→	in
Poss	→	my



The Dangling Else Problem

Who owns the *else*?

```
if (a) if (b) c(); else d();
```

Should this be

```
      if
     /  \
    a    if
         / | \
        b c() d()
```

or

```
      if
     /  |  \
    a  if  d()
         / \
        b  c()
```

?

Grammars are usually ambiguous; manuals give disambiguating rules such as C's:

As usual the "else" is resolved by connecting an else with the last encountered elseless if.

The Dangling Else Problem

```
stmt : IF expr THEN stmt  
      | IF expr THEN stmt ELSE stmt
```

Problem comes after matching the first statement. Question is whether an “else” should be part of the current statement or a surrounding one since the second line tells us “*stmt ELSE*” is possible.

The Dangling Else Problem

Some languages resolve this problem by insisting on nesting everything.

E.g., Algol 68:

```
if a < b then a else b fi;
```

“fi” is “if” spelled backwards. The language also uses do–od and case–esac.

Another Solution to the Dangling Else Problem

Idea: break into two types of statements: those that have a dangling “then” (“dstmt”) and those that do not (“cstmt”). A statement may be either, but the statement just before an “else” must not have a dangling clause because if it did, the “else” would belong to it.

```
stmt : dstmt
      | cstmt

dstmt : IF expr THEN stmt
      | IF expr THEN cstmt ELSE dstmt

cstmt : IF expr THEN cstmt ELSE cstmt
      | other statements...
```

We are effectively carrying an extra bit of information during parsing: whether there is an open “then” clause. Unfortunately, duplicating rules is the only way to do this in a context-free grammar.

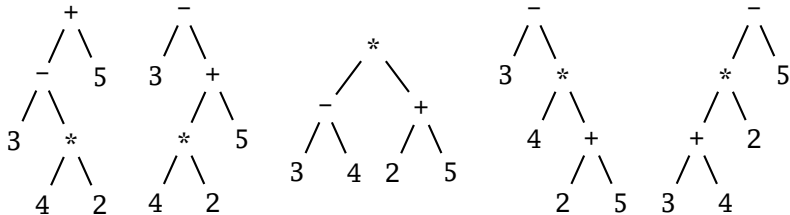
Ambiguous Arithmetic

Ambiguity can be a problem in expressions. Consider parsing

$$3 - 4 * 2 + 5$$

with the grammar

$$e \rightarrow e + e \mid e - e \mid e * e \mid e / e \mid N$$



Operator Precedence and Associativity

Usually resolve ambiguity in arithmetic expressions

Like you were taught in elementary school:

“My Dear Aunt Sally”

Mnemonic for multiplication and division before addition and subtraction.

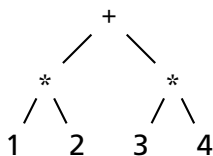
Operator Precedence

Defines how "sticky" an operator is.

$$1 * 2 + 3 * 4$$

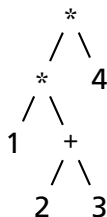
* at higher precedence than +:

$$(1 * 2) + (3 * 4)$$



+ at higher precedence than *:

$$1 * (2 + 3) * 4$$

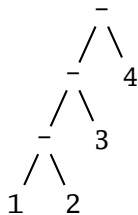


Associativity

Whether to evaluate left-to-right or right-to-left

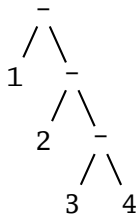
Most operators are left-associative

$$1 - 2 - 3 - 4$$



$$((1 - 2) - 3) - 4$$

left associative



$$1 - (2 - (3 - 4))$$

right associative

Fixing Ambiguous Grammars

A grammar specification:

```
expr :  
    expr PLUS expr  
    | expr MINUS expr  
    | expr TIMES expr  
    | expr DIVIDE expr  
    | NUMBER
```

Ambiguous: no precedence or associativity.

Ocamlyacc's complaint: "16 shift/reduce conflicts."

Assigning Precedence Levels

Split into multiple rules, one per level

```
expr : expr PLUS expr  
      | expr MINUS expr  
      | term  
  
term : term TIMES term  
      | term DIVIDE term  
      | atom  
  
atom : NUMBER
```

Still ambiguous: associativity not defined

Ocamlyacc's complaint: "8 shift/reduce conflicts."

Assigning Associativity

Make one side the next level of precedence

```
expr : expr PLUS term  
      | expr MINUS term  
      | term  
  
term : term TIMES atom  
      | term DIVIDE atom  
      | atom  
  
atom : NUMBER
```

This is left-associative.

No shift/reduce conflicts.

Statement separators/terminators

C uses ; as a statement terminator.

```
if (a<b)
    printf("a less");
else {
    printf("b"); printf(" less");
}
```

Pascal uses ; as a statement separator.

```
if a < b then
    writeln('a less')
else begin
    write('a'); writeln(' less')
end
```

Pascal later made a final ; optional.

Ocaml yacc Specifications

```
%{  
  (* Header: verbatim OCaml; optional *)  
%}  
  
/* Declarations: tokens, precedence, etc. */  
  
%%  
  
/* Rules: context-free rules */  
  
%%  
  
(* Trailer: verbatim OCaml; optional *)
```

Declarations

- ▶ `%token symbol ...`
Define symbol names (exported to .mli file)
- ▶ `%token < type > symbol ...`
Define symbols with attached attribute (also exported)
- ▶ `%start symbol ...`
Define start symbols (entry points)
- ▶ `%type < type > symbol ...`
Define the type for a symbol (mandatory for start)
- ▶ `%left symbol ...`
- ▶ `%right symbol ...`
- ▶ `%nonassoc symbol ...`
Define precedence and associativity for the given symbols, listed in order from lowest to highest precedence

Rules

```
nonterminal :  
  symbol ... symbol { semantic-action }  
  |  
  ...  
  | symbol ... symbol { semantic-action }
```

- ▶ *nonterminal* is the name of a rule, e.g., “program,” “expr”
- ▶ *symbol* is either a terminal (token) or another rule
- ▶ *semantic-action* is OCaml code evaluated when the rule is matched
- ▶ In a *semantic-action*, \$1, \$2, ... returns the value of the first, second, ... symbol matched
- ▶ A rule may include “%prec *symbol*” to override its default precedence

An Example .mly File

```
%token <int> INT
%token PLUS MINUS TIMES DIV LPAREN RPAREN EOL

%left PLUS MINUS /* lowest precedence */
%left TIMES DIV
%nonassoc UMINUS /* highest precedence */

%start main /* the entry point */
%type <int> main

%%

main:
    expr EOL                { $1 }

expr:
    INT                    { $1 }
  | LPAREN expr RPAREN    { $2 }
  | expr PLUS expr        { $1 + $3 }
  | expr MINUS expr       { $1 - $3 }
  | expr TIMES expr       { $1 * $3 }
  | expr DIV expr         { $1 / $3 }
  | MINUS expr %prec UMINUS { - $2 }
```

Parsing Algorithms

Parsing Context-Free Grammars

There are $O(n^3)$ algorithms for parsing arbitrary CFGs, but most compilers demand $O(n)$ algorithms.

Fortunately, the LL and LR subclasses of CFGs have $O(n)$ parsing algorithms. People use these in practice.

Rightmost Derivation of $\text{Id} * \text{Id} + \text{Id}$

e

1: $e \rightarrow t + e$

2: $e \rightarrow t$

3: $t \rightarrow \text{Id} * t$

4: $t \rightarrow \text{Id}$

At each step, expand the *rightmost* nonterminal.

nonterminal

“handle”: The right side of a production

Fun and interesting fact: there is exactly one rightmost expansion if the grammar is unambiguous.

Rightmost Derivation of $\text{Id} * \text{Id} + \text{Id}$

1: $e \rightarrow t + e$

2: $e \rightarrow t$

3: $t \rightarrow \text{Id} * t$

4: $t \rightarrow \text{Id}$

e
 $t + e$

At each step, expand the *rightmost* nonterminal.

nonterminal

“handle”: The right side of a production

Fun and interesting fact: there is exactly one rightmost expansion if the grammar is unambiguous.

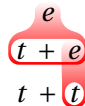
Rightmost Derivation of $\text{Id} * \text{Id} + \text{Id}$

1: $e \rightarrow t + e$

2: $e \rightarrow t$

3: $t \rightarrow \text{Id} * t$

4: $t \rightarrow \text{Id}$



At each step, expand the *rightmost* nonterminal.

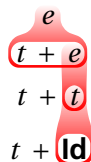
nonterminal

“handle”: The right side of a production

Fun and interesting fact: there is exactly one rightmost expansion if the grammar is unambiguous.

Rightmost Derivation of $\text{Id} * \text{Id} + \text{Id}$

- 1: $e \rightarrow t + e$
- 2: $e \rightarrow t$
- 3: $t \rightarrow \text{Id} * t$
- 4: $t \rightarrow \text{Id}$



At each step, expand the *rightmost* nonterminal.

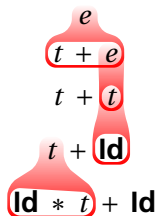
nonterminal

“handle”: The right side of a production

Fun and interesting fact: there is exactly one rightmost expansion if the grammar is unambiguous.

Rightmost Derivation of $\text{Id} * \text{Id} + \text{Id}$

- 1: $e \rightarrow t + e$
- 2: $e \rightarrow t$
- 3: $t \rightarrow \text{Id} * t$
- 4: $t \rightarrow \text{Id}$



At each step, expand the *rightmost* nonterminal.

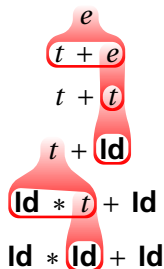
nonterminal

“handle”: The right side of a production

Fun and interesting fact: there is exactly one rightmost expansion if the grammar is unambiguous.

Rightmost Derivation of $\text{Id} * \text{Id} + \text{Id}$

- 1: $e \rightarrow t + e$
- 2: $e \rightarrow t$
- 3: $t \rightarrow \text{Id} * t$
- 4: $t \rightarrow \text{Id}$



At each step, expand the *rightmost* nonterminal.

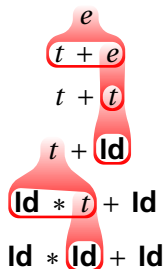
nonterminal

“handle”: The right side of a production

Fun and interesting fact: there is exactly one rightmost expansion if the grammar is unambiguous.

Rightmost Derivation of $\text{Id} * \text{Id} + \text{Id}$

- 1: $e \rightarrow t + e$
- 2: $e \rightarrow t$
- 3: $t \rightarrow \text{Id} * t$
- 4: $t \rightarrow \text{Id}$



At each step, expand the *rightmost* nonterminal.

nonterminal

“handle”: The right side of a production

Dragon-book style: underline handles

$e \rightarrow \underline{t + e} \rightarrow t + \underline{t} \rightarrow t + \underline{\text{Id}} \rightarrow \underline{\text{Id} * t} + \text{Id} \rightarrow \text{Id} * \underline{\text{Id}} + \text{Id}$

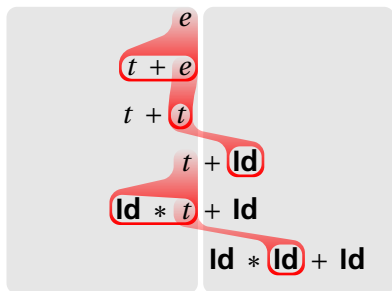
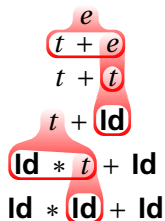
Rightmost Derivation: What to Expand

1: $e \rightarrow t + e$

2: $e \rightarrow t$

3: $t \rightarrow \text{Id} * t$

4: $t \rightarrow \text{Id}$



Expand here ↑

Terminals only

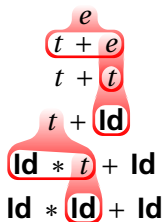
Reverse Rightmost Derivation

1: $e \rightarrow t + e$

2: $e \rightarrow t$

3: $t \rightarrow \mathbf{Id} * t$

4: $t \rightarrow \mathbf{Id}$



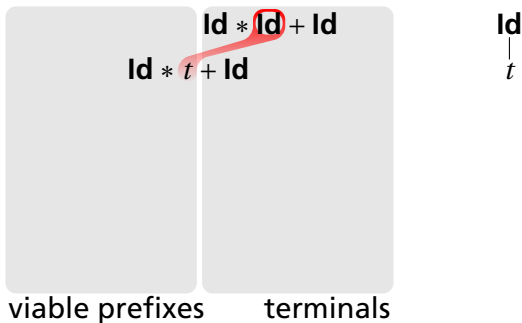
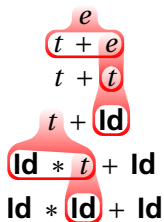
Reverse Rightmost Derivation

1: $e \rightarrow t + e$

2: $e \rightarrow t$

3: $t \rightarrow \mathbf{Id} * t$

4: $t \rightarrow \mathbf{Id}$



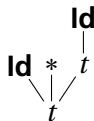
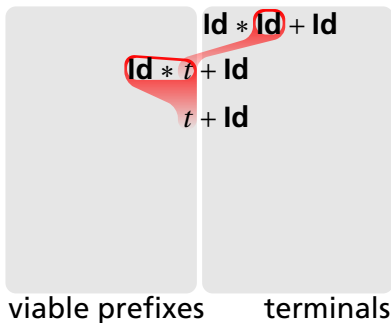
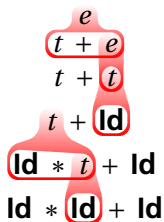
Reverse Rightmost Derivation

1: $e \rightarrow t + e$

2: $e \rightarrow t$

3: $t \rightarrow \text{Id} * t$

4: $t \rightarrow \text{Id}$



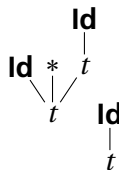
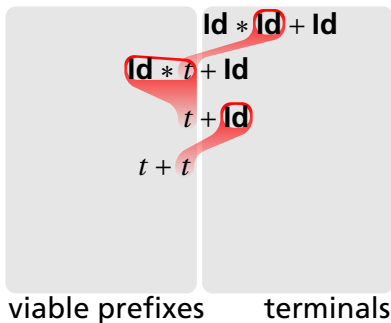
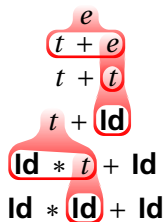
Reverse Rightmost Derivation

1: $e \rightarrow t + e$

2: $e \rightarrow t$

3: $t \rightarrow \text{Id} * t$

4: $t \rightarrow \text{Id}$



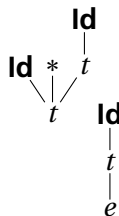
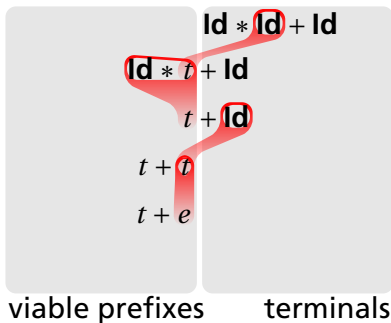
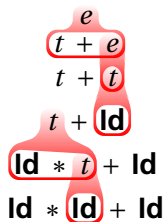
Reverse Rightmost Derivation

1: $e \rightarrow t + e$

2: $e \rightarrow t$

3: $t \rightarrow \text{Id} * t$

4: $t \rightarrow \text{Id}$



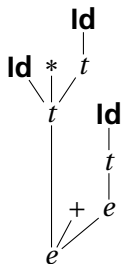
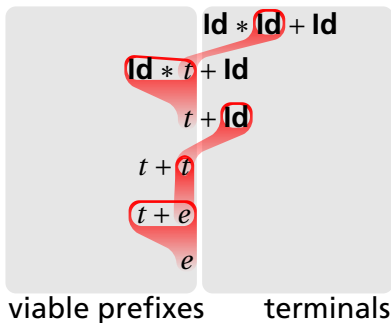
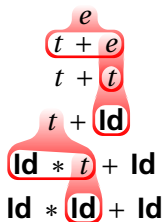
Reverse Rightmost Derivation

1: $e \rightarrow t + e$

2: $e \rightarrow t$

3: $t \rightarrow \text{Id} * t$

4: $t \rightarrow \text{Id}$



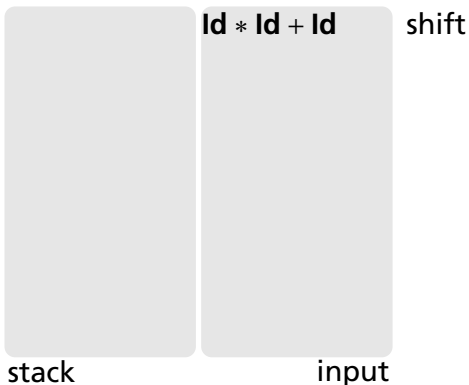
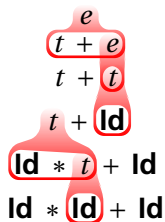
Shift/Reduce Parsing Using an Oracle

1: $e \rightarrow t + e$

2: $e \rightarrow t$

3: $t \rightarrow \mathbf{Id} * t$

4: $t \rightarrow \mathbf{Id}$



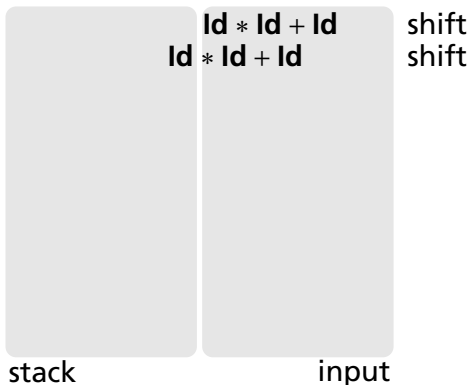
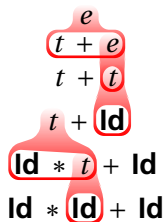
Shift/Reduce Parsing Using an Oracle

1: $e \rightarrow t + e$

2: $e \rightarrow t$

3: $t \rightarrow \mathbf{Id} * t$

4: $t \rightarrow \mathbf{Id}$



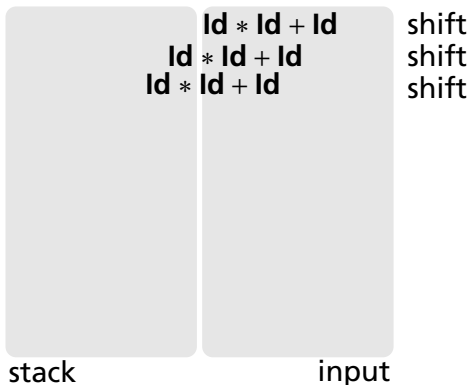
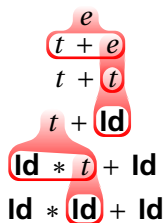
Shift/Reduce Parsing Using an Oracle

1: $e \rightarrow t + e$

2: $e \rightarrow t$

3: $t \rightarrow \mathbf{Id} * t$

4: $t \rightarrow \mathbf{Id}$



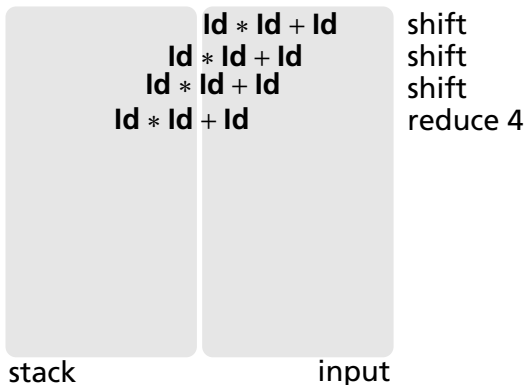
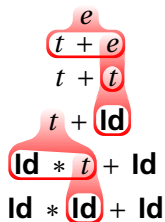
Shift/Reduce Parsing Using an Oracle

1: $e \rightarrow t + e$

2: $e \rightarrow t$

3: $t \rightarrow \mathbf{Id} * t$

4: $t \rightarrow \mathbf{Id}$



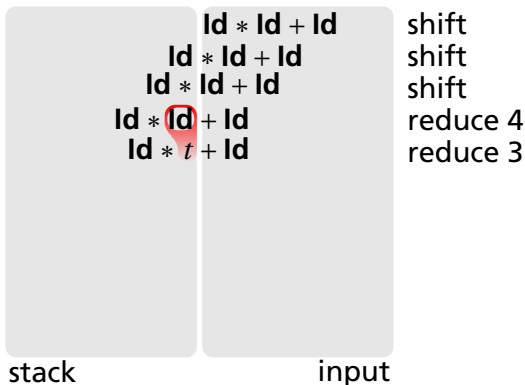
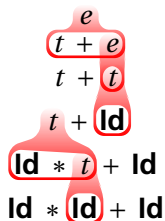
Shift/Reduce Parsing Using an Oracle

1: $e \rightarrow t + e$

2: $e \rightarrow t$

3: $t \rightarrow \mathbf{Id} * t$

4: $t \rightarrow \mathbf{Id}$



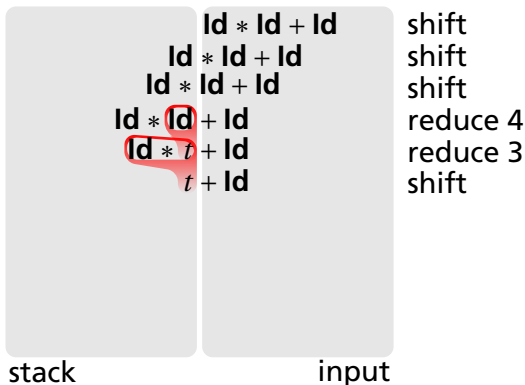
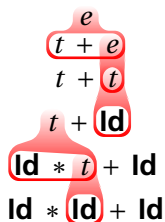
Shift/Reduce Parsing Using an Oracle

1: $e \rightarrow t + e$

2: $e \rightarrow t$

3: $t \rightarrow \mathbf{Id} * t$

4: $t \rightarrow \mathbf{Id}$



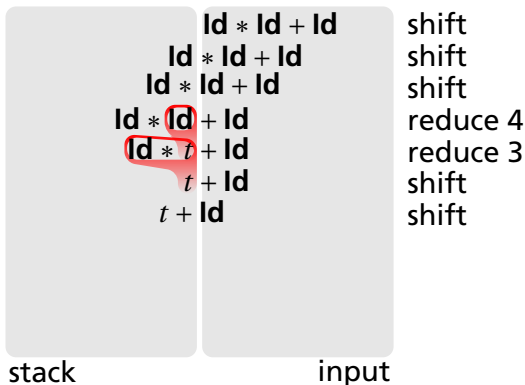
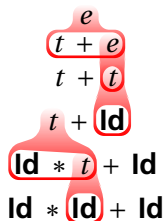
Shift/Reduce Parsing Using an Oracle

1: $e \rightarrow t + e$

2: $e \rightarrow t$

3: $t \rightarrow \text{Id} * t$

4: $t \rightarrow \text{Id}$



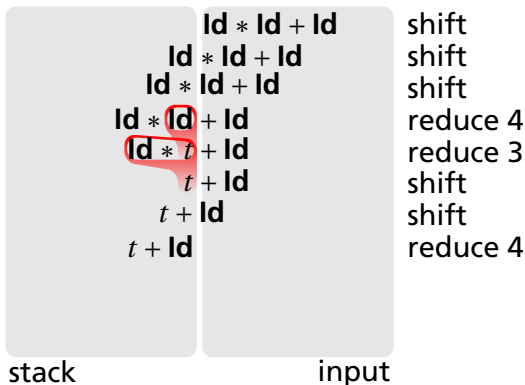
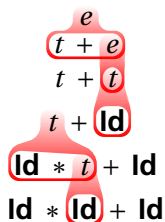
Shift/Reduce Parsing Using an Oracle

1: $e \rightarrow t + e$

2: $e \rightarrow t$

3: $t \rightarrow \text{Id} * t$

4: $t \rightarrow \text{Id}$



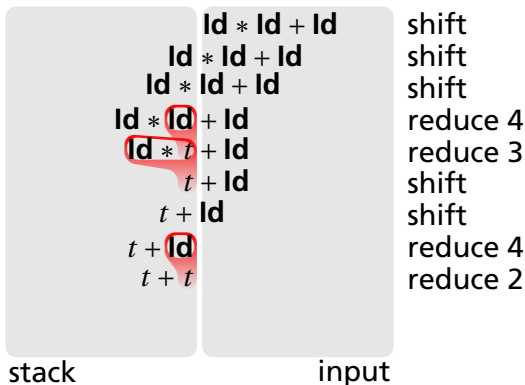
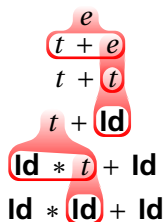
Shift/Reduce Parsing Using an Oracle

1: $e \rightarrow t + e$

2: $e \rightarrow t$

3: $t \rightarrow \text{Id} * t$

4: $t \rightarrow \text{Id}$



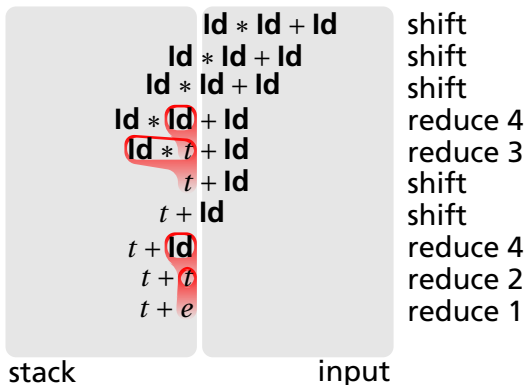
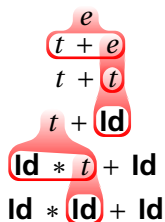
Shift/Reduce Parsing Using an Oracle

1: $e \rightarrow t + e$

2: $e \rightarrow t$

3: $t \rightarrow \text{Id} * t$

4: $t \rightarrow \text{Id}$



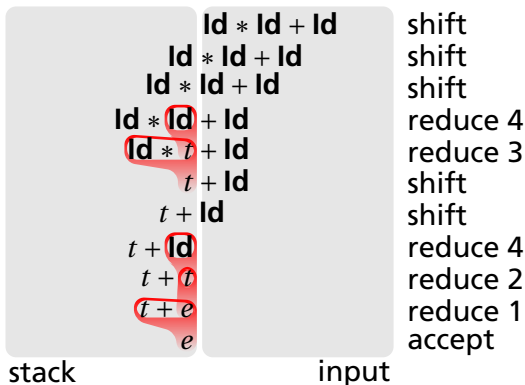
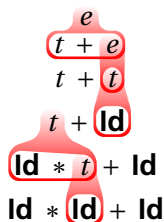
Shift/Reduce Parsing Using an Oracle

1: $e \rightarrow t + e$

2: $e \rightarrow t$

3: $t \rightarrow \text{Id} * t$

4: $t \rightarrow \text{Id}$



Handle Hunting

Right Sentential Form: any step in a rightmost derivation

Handle: in a sentential form, a RHS of a rule that, when rewritten, yields the previous step in a rightmost derivation.

The big question in shift/reduce parsing:

When is there a handle on the top of the stack?

Enumerate all the right-sentential forms and pattern-match against them? *Usually infinitely many; let's try anyway.*

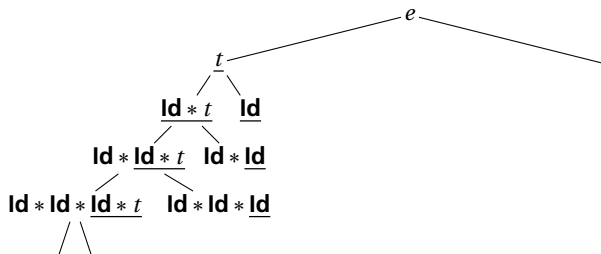
Some Right-Sentential Forms and Their Handles

1: $e \rightarrow t + e$

2: $e \rightarrow t$

3: $t \rightarrow \mathbf{ld} * t$

4: $t \rightarrow \mathbf{ld}$



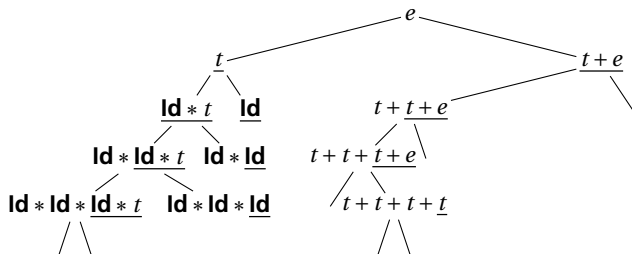
Some Right-Sentential Forms and Their Handles

1: $e \rightarrow t + e$

2: $e \rightarrow t$

3: $t \rightarrow \mathbf{ld} * t$

4: $t \rightarrow \mathbf{ld}$



Some Right-Sentential Forms and Their Handles

1: $e \rightarrow t + e$

2: $e \rightarrow t$

3: $t \rightarrow \mathbf{Id} * t$

4: $t \rightarrow \mathbf{Id}$

Patterns: $\mathbf{Id} * \mathbf{Id} * \dots * \underline{\mathbf{Id} * t} \dots$

$\mathbf{Id} * \mathbf{Id} * \dots * \underline{\mathbf{Id}} \dots$

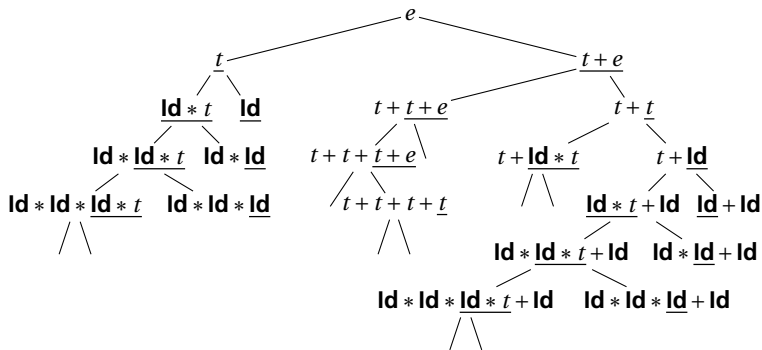
$t + t + \dots + \underline{t + e}$

$t + t + \dots + t + \underline{\mathbf{Id}}$

$t + t + \dots + t + \mathbf{Id} * \mathbf{Id} * \dots * \underline{\mathbf{Id} * t}$

$t + t + \dots + \underline{t}$

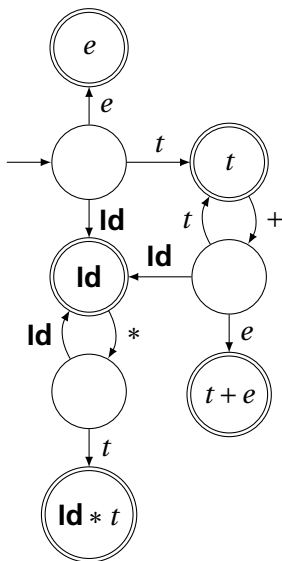
e



The Handle-Identifying Automaton

Magical result, due to Knuth: *An automaton suffices to locate a handle in a right-sentential form.*

$\text{ld} * \text{ld} * \dots * \underline{\text{ld}} * t \dots$
 $\text{ld} * \text{ld} * \dots * \underline{\text{ld}} \dots$
 $t + t + \dots + \underline{t + e}$
 $t + t + \dots + t + \underline{\text{ld}}$
 $t + t + \dots + t + \text{ld} * \text{ld} * \dots * \underline{\text{ld}} * t$
 $t + t + \dots + \underline{t}$
 e



Building the Initial State of the LR(0) Automaton

1: $e \rightarrow t + e$

2: $e \rightarrow t$

3: $t \rightarrow \mathbf{Id} * t$

4: $t \rightarrow \mathbf{Id}$

$$e' \rightarrow \color{yellow}{\curvearrowright} e$$

Key idea: automata identify viable prefixes of right sentential forms. Each state is an equivalence class of possible places in productions.

At the beginning, any viable prefix must be at the beginning of a string expanded from e . We write this condition " $e' \rightarrow \color{yellow}{\curvearrowright} e$ "

Building the Initial State of the LR(0) Automaton

1: $e \rightarrow t + e$

2: $e \rightarrow t$

3: $t \rightarrow \mathbf{Id} * t$

4: $t \rightarrow \mathbf{Id}$

$$e' \rightarrow \color{red}{\curvearrowright} e$$
$$e \rightarrow \color{red}{\curvearrowright} t + e$$
$$e \rightarrow \color{red}{\curvearrowright} t$$

Key idea: automata identify viable prefixes of right sentential forms. Each state is an equivalence class of possible places in productions.

At the beginning, any viable prefix must be at the beginning of a string expanded from e . We write this condition " $e' \rightarrow \color{red}{\curvearrowright} e$ "

There are two choices for what an e may expand to: $t + e$ and t . So when $e' \rightarrow \color{red}{\curvearrowright} e$, $e \rightarrow \color{red}{\curvearrowright} t + e$ and $e \rightarrow \color{red}{\curvearrowright} t$ are also true, i.e., it must start with a string expanded from t .

Building the Initial State of the LR(0) Automaton

- 1: $e \rightarrow t + e$
- 2: $e \rightarrow t$
- 3: $t \rightarrow \mathbf{ld} * t$
- 4: $t \rightarrow \mathbf{ld}$

$$\begin{array}{l} e' \rightarrow \mathbf{C}e \\ e \rightarrow \mathbf{C}t + e \\ e \rightarrow \mathbf{C}t \\ t \rightarrow \mathbf{Cld} * t \\ t \rightarrow \mathbf{Cld} \end{array}$$

Key idea: automata identify viable prefixes of right sentential forms. Each state is an equivalence class of possible places in productions.

At the beginning, any viable prefix must be at the beginning of a string expanded from e . We write this condition " $e' \rightarrow \mathbf{C}e$ "

There are two choices for what an e may expand to: $t + e$ and t . So when $e' \rightarrow \mathbf{C}e$, $e \rightarrow \mathbf{C}t + e$ and $e \rightarrow \mathbf{C}t$ are also true, i.e., it must start with a string expanded from t .

Also, t must be $\mathbf{ld} * t$ or \mathbf{ld} , so $t \rightarrow \mathbf{Cld} * t$ and $t \rightarrow \mathbf{Cld}$.

This is a *closure*, like ϵ -closure in subset construction.

Building the LR(0) Automaton

$$e' \rightarrow \color{yellow}{\curvearrowright} e$$
$$e \rightarrow \color{yellow}{\curvearrowright} t + e$$

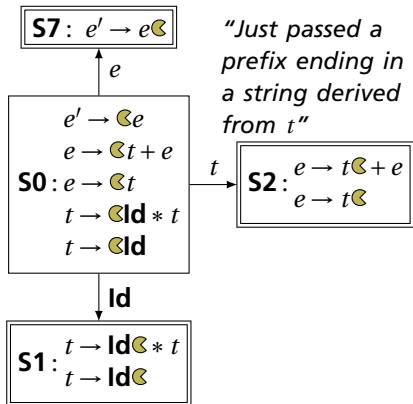
S0: $e \rightarrow \color{yellow}{\curvearrowright} t$

$$t \rightarrow \color{yellow}{\curvearrowright} \text{Id} * t$$
$$t \rightarrow \color{yellow}{\curvearrowright} \text{Id}$$

The first state suggests a viable prefix can start as any string derived from e , any string derived from t , or Id .

Building the LR(0) Automaton

"Just passed a string derived from e "



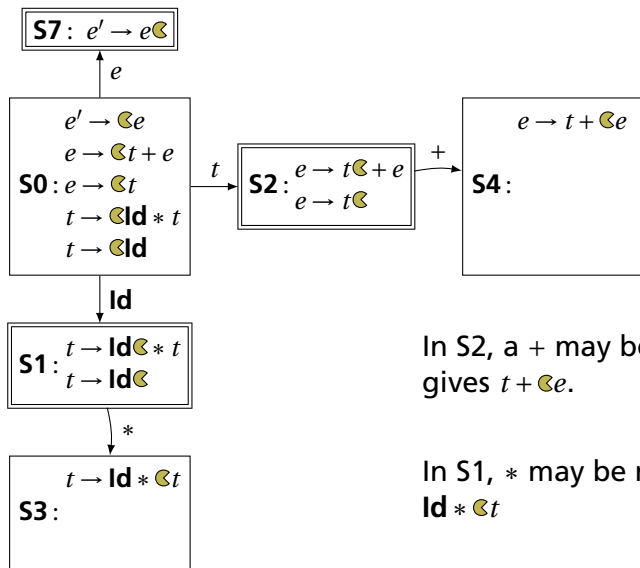
"Just passed a prefix that ended in an Id "

"Just passed a prefix ending in a string derived from t "

The first state suggests a viable prefix can start as any string derived from e , any string derived from t , or Id .

The items for these three states come from advancing the \bullet across each thing, then performing the closure operation (vacuous here).

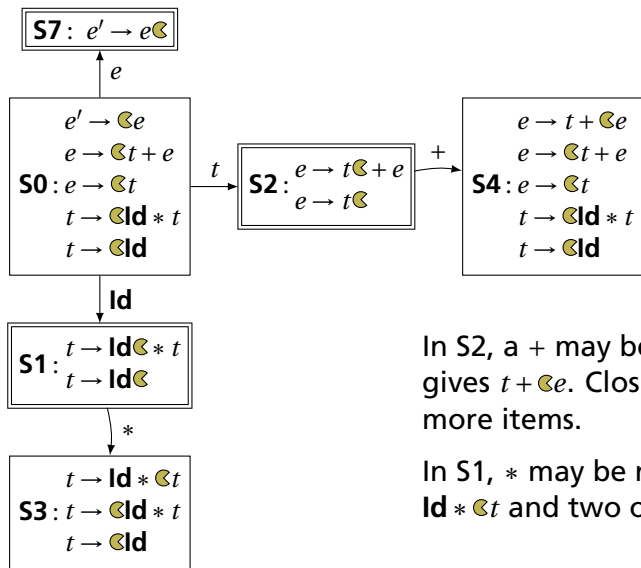
Building the LR(0) Automaton



In S2, a + may be next. This gives $t + \bullet e$.

In S1, * may be next, giving $\text{Id} * \bullet t$

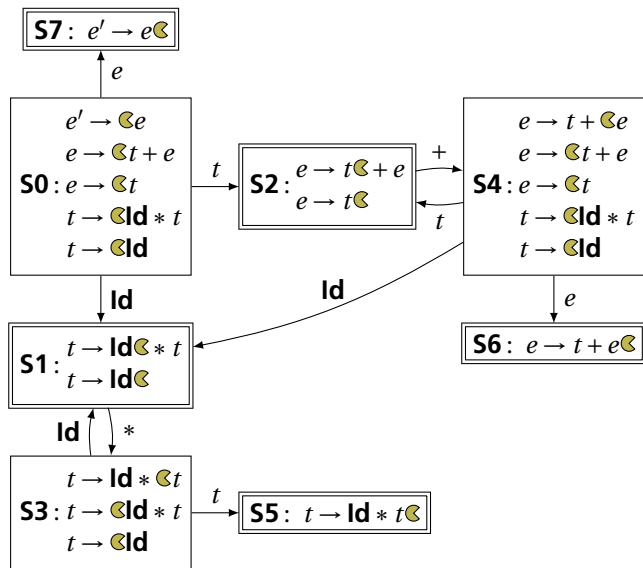
Building the LR(0) Automaton



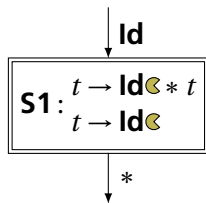
In S2, a + may be next. This gives $t + \bullet e$. Closure adds 4 more items.

In S1, * may be next, giving $\text{Id} * \bullet t$ and two others.

Building the LR(0) Automaton



What to do in each state?



1: $e \rightarrow t + e$

2: $e \rightarrow t$

3: $t \rightarrow \mathbf{Id} * t$

4: $t \rightarrow \mathbf{Id}$

$\mathbf{Id} * \mathbf{Id} * \dots * \mathbf{Id} * t \dots$

$\mathbf{Id} * \mathbf{Id} * \dots * \mathbf{Id} \dots$

$t + t + \dots + \underline{t + e}$

$t + t + \dots + t + \underline{\mathbf{Id}}$

$t + t + \dots + t + \mathbf{Id} * \mathbf{Id} * \dots * \mathbf{Id} * t$

$t + t + \dots + \underline{t}$

e

Stack	Input	Action
$\mathbf{Id} * \mathbf{Id} * \dots * \mathbf{Id}$	*...	Shift
$\mathbf{Id} * \mathbf{Id} * \dots * \mathbf{Id}$	+...	Reduce 4
$\mathbf{Id} * \mathbf{Id} * \dots * \mathbf{Id}$		Reduce 4
$\mathbf{Id} * \mathbf{Id} * \dots * \mathbf{Id}$	$\mathbf{Id} \dots$	Syntax Error

The first function

If you can derive a string that starts with terminal t from a sequence of terminals and nonterminals α , then $t \in \text{first}(\alpha)$.

1. If X is a terminal, $\text{first}(X) = \{X\}$.
2. If $X \rightarrow \epsilon$, then add ϵ to $\text{first}(X)$.
3. If $X \rightarrow Y_1 \cdots Y_k$ and $\epsilon \in \text{first}(Y_1)$, $\epsilon \in \text{first}(Y_2)$, \dots , and $\epsilon \in \text{first}(Y_{i-1})$ for $i = 1, \dots, k$ for some k ,
add $\text{first}(Y_i) - \{\epsilon\}$ to $\text{first}(X)$

X starts with anything that appears after skipping empty strings. Usually just $\text{first}(Y_1) \in \text{first}(X)$

4. If $X \rightarrow Y_1 \cdots Y_k$ and $\epsilon \in \text{first}(Y_1)$, $\epsilon \in \text{first}(Y_2)$, \dots , and $\epsilon \in \text{first}(Y_k)$, add ϵ to $\text{first}(X)$

If all of X can be empty, X can be empty

1: $e \rightarrow t + e$

$\text{first}(\mathbf{Id}) = \{\mathbf{Id}\}$

2: $e \rightarrow t$

$\text{first}(t) = \{\mathbf{Id}\}$ because $t \rightarrow \mathbf{Id} * t$ and $t \rightarrow \mathbf{Id}$

3: $t \rightarrow \mathbf{Id} * t$

$\text{first}(e) = \{\mathbf{Id}\}$ because $e \rightarrow t + e$, $e \rightarrow t$, and

4: $t \rightarrow \mathbf{Id}$

$\text{first}(t) = \{\mathbf{Id}\}$.

First and ϵ

$\epsilon \in \text{first}(\alpha)$ means α can derive the empty string.

1. If X is a terminal, $\text{first}(X) = \{X\}$.
2. If $X \rightarrow \epsilon$, then add ϵ to $\text{first}(X)$.
3. If $X \rightarrow Y_1 \cdots Y_k$ and
 $\epsilon \in \text{first}(Y_1)$, $\epsilon \in \text{first}(Y_2)$, \dots , and $\epsilon \in \text{first}(Y_{i-1})$
for $i = 1, \dots, k$ for some k ,
add $\text{first}(Y_i) - \{\epsilon\}$ to $\text{first}(X)$
4. If $X \rightarrow Y_1 \cdots Y_K$ and
 $\epsilon \in \text{first}(Y_1)$, $\epsilon \in \text{first}(Y_2)$, \dots , and $\epsilon \in \text{first}(Y_k)$,
add ϵ to $\text{first}(X)$

$X \rightarrow YZa$	$\text{first}(b) = \{b\}$ $\text{first}(c) = \{c\}$ $\text{first}(d) = \{d\}$	(1)
$Y \rightarrow$	$\text{first}(W) = \{\epsilon\} \cup \text{first}(d) = \{\epsilon, d\}$	(2, 3)
$Y \rightarrow b$	$\text{first}(Z) = \text{first}(c) \cup (\text{first}(W) - \{\epsilon\}) \cup \{\epsilon\} = \{\epsilon, c, d\}$	(3, 3, 4)
$Z \rightarrow c$	$\text{first}(Y) = \{\epsilon\} \cup \{b\} = \{\epsilon, b\}$	(2, 3)
$Z \rightarrow W$	$\text{first}(X) = (\text{first}(Y) - \{\epsilon\}) \cup (\text{first}(Z) - \{\epsilon\}) \cup$	
$W \rightarrow$	$\text{first}(a) = \{a, b, c, d\}$	(3, 3, 3)
$W \rightarrow d$		

The follow function

If t is a terminal, A is a nonterminal, and $\dots At\dots$ can be derived, then $t \in \text{follow}(A)$.

1. Add \$ ("end-of-input") to $\text{follow}(S)$ (start symbol).
End-of-input comes after the start symbol
2. For each prod. $\rightarrow \dots A\alpha$, add $\text{first}(\alpha) - \{\epsilon\}$ to $\text{follow}(A)$.
A is followed by the first thing after it
3. For each prod. $A \rightarrow \dots B$ or $A \rightarrow \dots B\alpha$ where $\epsilon \in \text{first}(\alpha)$, then add everything in $\text{follow}(A)$ to $\text{follow}(B)$.
If B appears at the end of a production, it can be followed by whatever follows that production

1: $e \rightarrow t + e$

$\text{follow}(e) = \{\$\}$

2: $e \rightarrow t$

$\text{follow}(t) = \{ \quad \}$

3: $t \rightarrow \mathbf{ld} * t$

1. Because e is the start symbol

4: $t \rightarrow \mathbf{ld}$

$\text{first}(t) = \{\mathbf{ld}\}$

$\text{first}(e) = \{\mathbf{ld}\}$

The follow function

If t is a terminal, A is a nonterminal, and $\dots At\dots$ can be derived, then $t \in \text{follow}(A)$.

1. Add \$ ("end-of-input") to $\text{follow}(S)$ (start symbol).
End-of-input comes after the start symbol
2. For each prod. $\rightarrow \dots A\alpha$, add $\text{first}(\alpha) - \{\epsilon\}$ to $\text{follow}(A)$.
A is followed by the first thing after it
3. For each prod. $A \rightarrow \dots B$ or $A \rightarrow \dots B\alpha$ where $\epsilon \in \text{first}(\alpha)$, then add everything in $\text{follow}(A)$ to $\text{follow}(B)$.
If B appears at the end of a production, it can be followed by whatever follows that production

1: $e \rightarrow t + e$

$\text{follow}(e) = \{\$\}$

2: $e \rightarrow t$

$\text{follow}(t) = \{+ \}$

3: $t \rightarrow \mathbf{ld} * t$

2. Because $e \rightarrow \underline{t} + e$ and $\text{first}(+) = \{+\}$

4: $t \rightarrow \mathbf{ld}$

$\text{first}(t) = \{\mathbf{ld}\}$

$\text{first}(e) = \{\mathbf{ld}\}$

The follow function

If t is a terminal, A is a nonterminal, and $\dots At\dots$ can be derived, then $t \in \text{follow}(A)$.

1. Add \$ ("end-of-input") to $\text{follow}(S)$ (start symbol).
End-of-input comes after the start symbol
2. For each prod. $\rightarrow \dots A\alpha$, add $\text{first}(\alpha) - \{\epsilon\}$ to $\text{follow}(A)$.
A is followed by the first thing after it
3. For each prod. $A \rightarrow \dots B$ or $A \rightarrow \dots B\alpha$ where $\epsilon \in \text{first}(\alpha)$, then add everything in $\text{follow}(A)$ to $\text{follow}(B)$.
If B appears at the end of a production, it can be followed by whatever follows that production

1: $e \rightarrow t + e$

$\text{follow}(e) = \{\$\}$

2: $e \rightarrow t$

$\text{follow}(t) = \{+, \$\}$

3: $t \rightarrow \mathbf{ld} * t$

3. Because $e \rightarrow \underline{t}$ and $\$ \in \text{follow}(e)$

4: $t \rightarrow \mathbf{ld}$

$\text{first}(t) = \{\mathbf{ld}\}$

$\text{first}(e) = \{\mathbf{ld}\}$

The follow function

If t is a terminal, A is a nonterminal, and $\dots At\dots$ can be derived, then $t \in \text{follow}(A)$.

1. Add \$ ("end-of-input") to $\text{follow}(S)$ (start symbol).
End-of-input comes after the start symbol
2. For each prod. $\rightarrow \dots A\alpha$, add $\text{first}(\alpha) - \{\epsilon\}$ to $\text{follow}(A)$.
A is followed by the first thing after it
3. For each prod. $A \rightarrow \dots B$ or $A \rightarrow \dots B\alpha$ where $\epsilon \in \text{first}(\alpha)$, then add everything in $\text{follow}(A)$ to $\text{follow}(B)$.
If B appears at the end of a production, it can be followed by whatever follows that production

1: $e \rightarrow t + e$

$\text{follow}(e) = \{\$\}$

2: $e \rightarrow t$

$\text{follow}(t) = \{+, \$\}$

3: $t \rightarrow \mathbf{ld} * t$

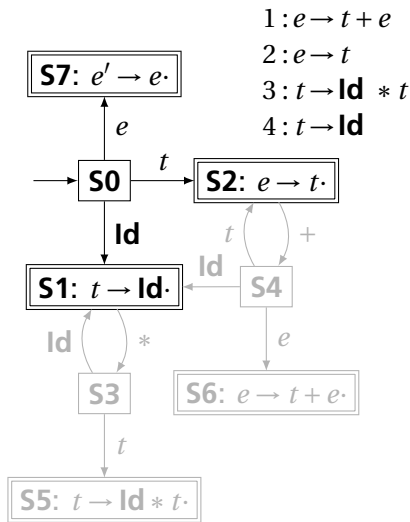
4: $t \rightarrow \mathbf{ld}$

$\text{first}(t) = \{\mathbf{ld}\}$

Fixed-point reached: applying any rule does not change any set

$\text{first}(e) = \{\mathbf{ld}\}$

Converting the LR(0) Automaton to an SLR Table



$\text{follow}(e) = \{\$\}$

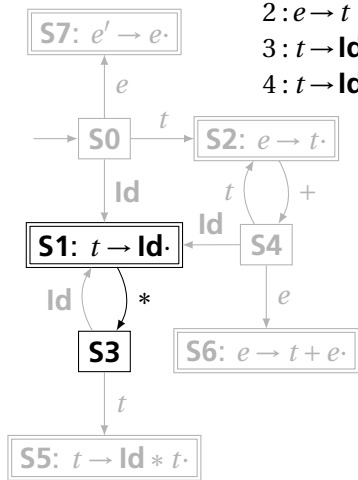
$\text{follow}(t) = \{+, \$\}$

State	Action				Goto	
	Id	+	*	\$	e	t
0	s1				7	2

From S_0 , shift an **Id** and go to S_1 ; or cross a t and go to S_2 ; or cross an e and go to S_7 .

Converting the LR(0) Automaton to an SLR Table

- 1: $e \rightarrow t + e$
 2: $e \rightarrow t$
 3: $t \rightarrow \text{Id} * t$
 4: $t \rightarrow \text{Id}$



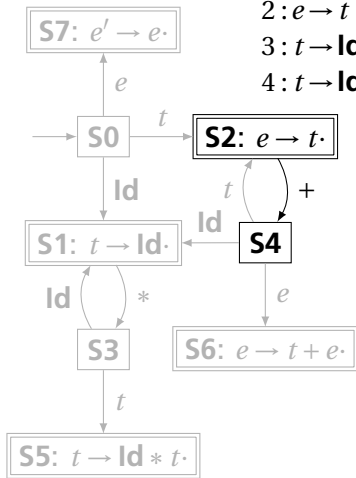
$\text{follow}(e) = \{\$\}$
 $\text{follow}(t) = \{+, \$\}$

State	Action				Goto	
	Id	+	*	\$	e	t
0	s1				7	2
1		r4	s3	r4		

From S1, shift a $*$ and go to S3;
 or, if the next input $\in \text{follow}(t)$,
 reduce by rule 4.

Converting the LR(0) Automaton to an SLR Table

- 1: $e \rightarrow t + e$
 2: $e \rightarrow t$
 3: $t \rightarrow \text{Id} * t$
 4: $t \rightarrow \text{Id}$



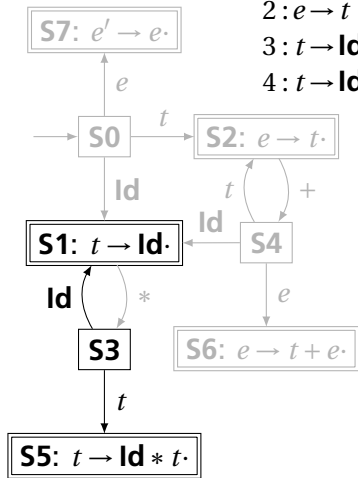
$\text{follow}(e) = \{\$ \}$
 $\text{follow}(t) = \{+, \$ \}$

State	Action				Goto	
	Id	+	*	\$	e	t
0	s1				7	2
1		r4	s3	r4		
2		s4		r2		

From S2, shift a + and go to S4;
 or, if the next input $\in \text{follow}(e)$,
 reduce by rule 2.

Converting the LR(0) Automaton to an SLR Table

- 1: $e \rightarrow t + e$
 2: $e \rightarrow t$
 3: $t \rightarrow \text{Id} * t$
 4: $t \rightarrow \text{Id}$



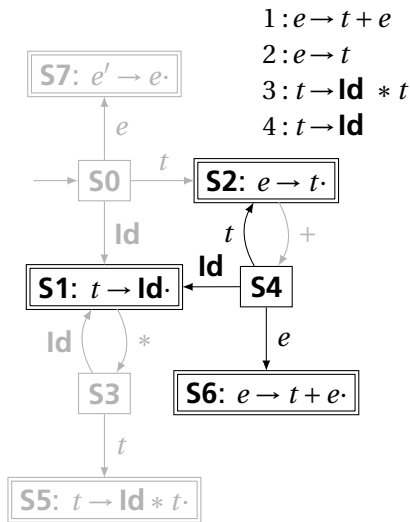
$\text{follow}(e) = \{\$ \}$

$\text{follow}(t) = \{+, \$ \}$

State	Action				Goto	
	Id	+	*	\$	e	t
0	s1				7	2
1		r4	s3	r4		
2		s4		r2		
3	s1					5

From S3, shift an **Id** and go to S1;
 or cross a **t** and go to S5.

Converting the LR(0) Automaton to an SLR Table



$\text{follow}(e) = \{\$\}$

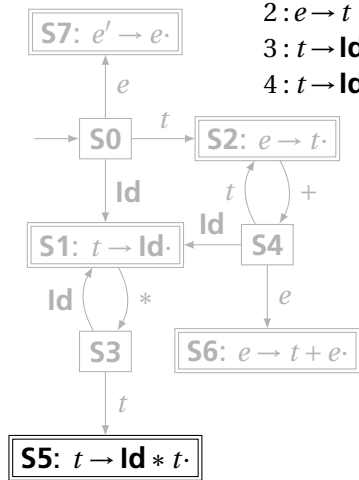
$\text{follow}(t) = \{+, \$\}$

State	Action				Goto	
	Id	+	*	\$	e	t
0	s1				7	2
1		r4	s3	r4		
2		s4		r2		
3	s1					5
4	s1				6	2

From S4, shift an **Id** and go to S1;
 or cross an *e* or a *t*.

Converting the LR(0) Automaton to an SLR Table

- 1: $e \rightarrow t + e$
 2: $e \rightarrow t$
 3: $t \rightarrow \text{Id} * t$
 4: $t \rightarrow \text{Id}$



$\text{follow}(e) = \{\$\}$

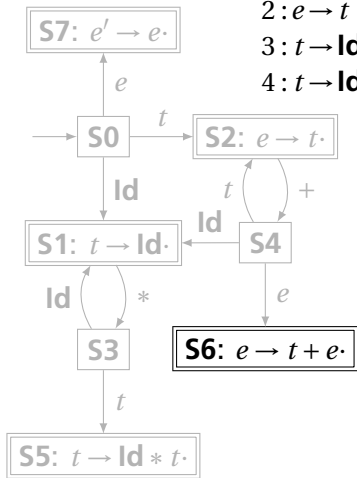
$\text{follow}(t) = \{+, \$\}$

State	Action				Goto	
	Id	+	*	\$	e	t
0	s1				7	2
1		r4	s3	r4		
2		s4		r2		
3	s1					5
4	s1				6	2
5		r3		r3		

From S5, reduce using rule 3 if the next symbol $\in \text{follow}(t)$.

Converting the LR(0) Automaton to an SLR Table

- 1: $e \rightarrow t + e$
 2: $e \rightarrow t$
 3: $t \rightarrow \text{Id} * t$
 4: $t \rightarrow \text{Id}$



$\text{follow}(e) = \{\$ \}$

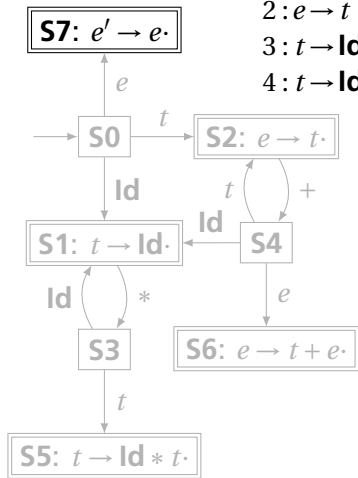
$\text{follow}(t) = \{+, \$ \}$

State	Action				Goto	
	Id	+	*	\$	e	t
0	s1				7	2
1		r4	s3	r4		
2		s4		r2		
3	s1					5
4	s1				6	2
5		r3		r3		
6				r1		

From S6, reduce using rule 1 if the next symbol $\in \text{follow}(e)$.

Converting the LR(0) Automaton to an SLR Table

- 1: $e \rightarrow t + e$
- 2: $e \rightarrow t$
- 3: $t \rightarrow \text{Id} * t$
- 4: $t \rightarrow \text{Id}$



$\text{follow}(e) = \{\$ \}$
 $\text{follow}(t) = \{+, \$ \}$

State	Action				Goto	
	Id	+	*	\$	e	t
0	s1				7	2
1		r4	s3	r4		
2		s4		r2		
3	s1					5
4	s1				6	2
5		r3		r3		
6				r1		
7				✓		

If, in S7, we just crossed an e , accept if we are at the end of the input.

Shift/Reduce Parsing with an SLR Table

1: $e \rightarrow t + e$

2: $e \rightarrow t$

3: $t \rightarrow \mathbf{Id} * t$

4: $t \rightarrow \mathbf{Id}$

State	Action				Goto	
	Id	+	*	\$	e	t
0	s1				7	2
1		r4	s3	r4		
2		s4		r2		
3	s1					5
4	s1				6	2
5		r3		r3		
6				r1		
7				✓		

Stack	Input	Action
0	Id * Id + Id \$	Shift, goto 1

Look at the state on top of the stack and the next input token.

Find the action (shift, reduce, or error) in the table.

In this case, shift the token onto the stack and mark it with state 1.

Shift/Reduce Parsing with an SLR Table

1: $e \rightarrow t + e$

2: $e \rightarrow t$

3: $t \rightarrow \mathbf{Id} * t$

4: $t \rightarrow \mathbf{Id}$

Stack	Input	Action
0	Id * Id + Id \$	Shift, goto 1
0 Id 1	* Id + Id \$	Shift, goto 3

Here, the state is 1, the next symbol is *, so shift and mark it with state 3.

State	Action				Goto	
	Id	+	*	\$	e	t
0	s1				7	2
1		r4	s3	r4		
2		s4		r2		
3	s1					5
4	s1				6	2
5		r3		r3		
6				r1		
7				✓		

Shift/Reduce Parsing with an SLR Table

1: $e \rightarrow t + e$

2: $e \rightarrow t$

3: $t \rightarrow \mathbf{Id} * t$

4: $t \rightarrow \mathbf{Id}$

State	Action				Goto	
	Id	+	*	\$	e	t
0	s1				7	2
1		r4	s3	r4		
2		s4		r2		
3	s1					5
4	s1				6	2
5		r3		r3		
6				r1		
7				✓		

Stack	Input	Action
		0
	Id * Id + Id \$	Shift, goto 1
		0 Id 1
	* Id + Id \$	Shift, goto 3
		0 Id * 1 3
	Id + Id \$	Shift, goto 1
0 Id 1 3 1	+ Id \$	Reduce 4

Here, the state is 1, the next symbol is +. The table says reduce using rule 4.

Shift/Reduce Parsing with an SLR Table

1: $e \rightarrow t + e$

2: $e \rightarrow t$

3: $t \rightarrow \text{Id} * t$

4: $t \rightarrow \text{Id}$

State	Action				Goto	
	Id	+	*	\$	e	t
0	s1				7	2
1		r4	s3	r4		
2		s4		r2		
3	s1					5
4	s1				6	2
5		r3		r3		
6				r1		
7				✓		

Stack	Input	Action
0	Id * Id + Id \$	Shift, goto 1
0 Id	* Id + Id \$	Shift, goto 3
0 Id *	Id + Id \$	Shift, goto 1
0 Id * Id	+ Id \$	Reduce 4
0 Id *	+ Id \$	

Remove the RHS of the rule (the handle: here, just **Id**), observe the state on the top of the stack, and consult the "goto" portion of the table.

Shift/Reduce Parsing with an SLR Table

- 1: $e \rightarrow t + e$
- 2: $e \rightarrow t$
- 3: $t \rightarrow \text{Id} * t$
- 4: $t \rightarrow \text{Id}$

State	Action				Goto	
	Id	+	*	\$	<i>e</i>	<i>t</i>
0	s1				7	2
1		r4	s3	r4		
2		s4		r2		
3	s1					5
4	s1				6	2
5		r3		r3		
6				r1		
7				✓		

Stack	Input	Action
0	Id * Id + Id \$	Shift, goto 1
0 Id	* Id + Id \$	Shift, goto 3
0 Id *	Id + Id \$	Shift, goto 1
0 Id * Id	+ Id \$	Reduce 4
0 Id * Id *	+ Id \$	Reduce 3

Here, we push a *t* with state 5. This effectively “backs up” the LR(0) automaton and runs it over the newly added nonterminal.

In state 5 with an upcoming +, the action is “reduce 3.”

Shift/Reduce Parsing with an SLR Table

1: $e \rightarrow t + e$

2: $e \rightarrow t$

3: $t \rightarrow \mathbf{Id} * t$

4: $t \rightarrow \mathbf{Id}$

State	Action				Goto	
	Id	+	*	\$	<i>e</i>	<i>t</i>
0	s1				7	2
1		r4	s3	r4		
2		s4		r2		
3	s1					5
4	s1				6	2
5		r3		r3		
6				r1		
7				✓		

Stack	Input	Action
0	Id * Id + Id \$	Shift, goto 1
0 Id	* Id + Id \$	Shift, goto 3
0 Id *	Id + Id \$	Shift, goto 1
0 Id * Id	+ Id \$	Reduce 4
0 Id * Id	+ Id \$	Reduce 3
0 t	+ Id \$	Shift, goto 4

This time, we strip off the RHS for rule 3, the handle $\mathbf{Id} * t$, exposing state 0, so we push a t with state 2.

Shift/Reduce Parsing with an SLR Table

1: $e \rightarrow t + e$

2: $e \rightarrow t$

3: $t \rightarrow \text{Id} * t$

4: $t \rightarrow \text{Id}$

State	Action				Goto	
	Id	+	*	\$	e	t
0	s1				7	2
1		r4	s3	r4		
2		s4		r2		
3	s1					5
4	s1				6	2
5		r3		r3		
6				r1		
7				✓		

Stack	Input	Action
0	Id * Id + Id \$	Shift, goto 1
0 Id	* Id + Id \$	Shift, goto 3
0 Id *	Id + Id \$	Shift, goto 1
0 Id * Id	+ Id \$	Reduce 4
0 Id * Id	+ Id \$	Reduce 3
0 t	+ Id \$	Shift, goto 4
0 t +	Id \$	Shift, goto 1
0 t + Id	\$	Reduce 4
0 t + t	\$	Reduce 2
0 t + e	\$	Reduce 1
0 e	\$	Accept

L, R, and all that

LR parser: "Bottom-up parser":

L = Left-to-right scan, R = (reverse) Rightmost derivation

RR parser: R = Right-to-left scan (from end)

I called them "Australian style"; nobody uses these

LL parser: "Top-down parser":

L = Left-to-right scan: L = (reverse) Leftmost derivation

LR(1): LR parser that considers next token (lookahead of 1)

LR(0): Only considers stack to decide shift/reduce

SLR(1): Simple LR: lookahead from first/follow rules

Derived from LR(0) automaton

LALR(1): Lookahead LR(1): fancier lookahead analysis

Uses same LR(0) automaton as SLR(1)

Ocamlyacc builds LALR(1) tables.

The Punchline

This is a tricky, but mechanical procedure. The Ocamlyacc parser generator uses a modified version of this technique to generate fast bottom-up parsers.

You need to understand it to comprehend error messages:

Shift/reduce conflicts are caused by a state like

$$t \rightarrow \cdot \mathbf{Else} s$$
$$t \rightarrow \cdot$$

If the next token is **Else**, do you reduce it since **Else** may follow a t , or shift it?

Reduce/reduce conflicts are caused by a state like

$$t \rightarrow \mathbf{Id} * t \cdot$$
$$e \rightarrow \mathbf{Id} * t \cdot$$

Do you reduce by " $t \rightarrow \mathbf{Id} * t$ " or by " $e \rightarrow \mathbf{Id} * t$ "?

A Reduce/Reduce Conflict

- 1: $a \rightarrow \text{Id Id}$
- 2: $a \rightarrow b$
- 3: $b \rightarrow \text{Id Id}$

